Introduction to Cryptography with Coding Theory, 3rd

awrence C. Washington
Wade Trappe

VISIT...

Introduction to Cryptography

with Coding Theory

3rd edition

Wade Trappe

Wireless Information Network Laboratory and the Electrical and Computer Engineering Department
Rutgers University

Lawrence C. Washington

Department of Mathematics University of Maryland

Portfolio Manager: *Chelsea Kharakozoua*

Content Manager: *Jeff Weidenaar*

Content Associate: *Jonathan Krebs*

Content Producer: *Tara Corpuz*

Managing Producer: *Scott Disanno*

Producer: *Jean Choe*

Manager, Courseware QA: *Mary Durnwald*

Product Marketing Manager: *Stacey Sveum*

Product and Solution Specialist: *Rosemary Morten*

Senior Author Support/Technology Specialist: *Joe Vetere*

Manager, Rights and Permissions: *Gina Cheselka*

Text and Cover Design, Production Coordination, Composition, and Illustrations: *Integra Software Services Pvt. Ltd*

Manufacturing Buyer: *Carol Melville, LSC Communications*

Cover Image: *Photographer is my life/ Getty Images*

Copyright © 2020, 2006, 2002 by Pearson Education, Inc. 221 River Street, Hoboken, NJ 07030. All Rights Reserved. Printed in the United States of America. This publication is protected by copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions department, please visit www.pearsoned.com/permissions/.

Text Credit: Page 23 Declaration of

Independence: A Transcription, The U.S. National Archives and Records Administration.

PEARSON, ALWAYS LEARNING, and MYLAB are exclusive trademarks owned by Pearson Education, Inc. or its affiliates in the U.S. and/or other countries.

Unless otherwise indicated herein, any third-party trademarks that may appear in this work are the property of their respective owners and any references to third-party trademarks, logos or other trade dress are for demonstrative or descriptive purposes only. Such references are not intended to imply any sponsorship, endorsement, authorization, or promotion of Pearson's

products by the owners of such marks, or any relationship between the owner and Pearson Education, Inc. or its affiliates, authors, licensees or distributors.

Library of Congress Cataloging-in-Publication Data

Names: Trappe, Wade, author. | Washington, Lawrence C., author.

Title: Introduction to cryptography : with coding theory / Wade Trappe, Lawrence Washington.

Description: 3rd edition. | [Hoboken, New Jersey] : [Pearson Education], [2020] | Includes bibliographical references and index. | Summary: “This

book is based on a course in cryptography at the upper-level undergraduate and beginning graduate level that has been given at the University of Maryland since 1997, and a course that has been taught at Rutgers University since 2003"— Provided by publisher.

Identifiers: LCCN 2019029691 | ISBN 9780134860992 (paperback)

Subjects: LCSH: Coding theory. | Cryptography.

Classification: LCC QA268.T73 2020 | DDC 005.8/24—dc23

LC record available at https://lccn.loc.gov/2019029691

ScoutAutomatedPrintCode

ISBN-13: 978-0-13-485906-4

ISBN-10: 0-13-485906-5

Contents

1. Preface ix

1. 1 Overview of Cryptography and Its Applications 1

    1. 1.1 Secure Communications 2

    2. 1.2 Cryptographic Applications 8

2. 2 Classical Cryptosystems 10

    1. 2.1 Shift Ciphers 11

    2. 2.2 Affine Ciphers 12

    3. 2.3 The Vigenère Cipher 14

    4. 2.4 Substitution Ciphers 20

    5. 2.5 Sherlock Holmes 23

    6. 2.6 The Playfair and ADFGX Ciphers 26

    7. 2.7 Enigma 29

8. 2.8 Exercises 33

9. 2.9 Computer Problems 37

3. 3 Basic Number Theory 40

1. 3.1 Basic Notions 40

2. 3.2 The Extended Euclidean Algorithm 44

3. 3.3 Congruences 47

4. 3.4 The Chinese Remainder Theorem 52

5. 3.5 Modular Exponentiation 54

6. 3.6 Fermat's Theorem and Euler's Theorem 55

7. 3.7 Primitive Roots 59

8. 3.8 Inverting Matrices Mod *n* 61

9. 3.9 Square Roots Mod *n* 62

10. 3.10 Legendre and Jacobi Symbols 64

11. 3.11 Finite Fields 69

12. 3.12 Continued Fractions 76

13. 3.13 Exercises 78

14. 3.14 Computer Problems 86

4. 4 The One-Time Pad 88

    1. 4.1 Binary Numbers and ASCII 88

    2. 4.2 One-Time Pads 89

    3. 4.3 Multiple Use of a One-Time Pad 91

    4. 4.4 Perfect Secrecy of the One-Time Pad 94

    5. 4.5 Indistinguishability and Security 97

    6. 4.6 Exercises 100

5. 5 Stream Ciphers 104

    1. 5.1 Pseudorandom Bit Generation 105

    2. 5.2 LFSR Sequences 107

    3. 5.3 RC4 113

    4. 5.4 Exercises 114

    5. 5.5 Computer Problems 117

6. 6 Block Ciphers 118

    1. 6.1 Block Ciphers 118

    2. 6.2 Hill Ciphers 119

    3. 6.3 Modes of Operation 122

    4. 6.4 Multiple Encryption 129

    5. 6.5 Meet-in-the-Middle Attacks 130

    6. 6.6 Exercises 131

    7. 6.7 Computer Problems 135

7. 7 The Data Encryption Standard 136

    1. 7.1 Introduction 136

    2. 7.2 A Simplified DES-Type Algorithm 137

    3. 7.3 Differential Cryptanalysis 140

    4. 7.4 DES 145

    5. 7.5 Breaking DES 152

    6. 7.6 Password Security 155

7. 7.7 Exercises 157

8. 7.8 Computer Problems 159

8. 8 The Advanced Encryption Standard: Rijndael 160

1. 8.1 The Basic Algorithm 160

2. 8.2 The Layers 161

3. 8.3 Decryption 166

4. 8.4 Design Considerations 168

5. 8.5 Exercises 169

9. 9 The RSA Algorithm 171

1. 9.1 The RSA Algorithm 171

2. 9.2 Attacks on RSA 177

3. 9.3 Primality Testing 183

4. 9.4 Factoring 188

5. 9.5 The RSA Challenge 192

6. 9.6 An Application to Treaty Verification 194

7. 9.7 The Public Key Concept 195

8. 9.8 Exercises 197

9. 9.9 Computer Problems 207

10. 10 Discrete Logarithms 211

    1. 10.1 Discrete Logarithms 211

    2. 10.2 Computing Discrete Logs 212

    3. 10.3 Bit Commitment 218

    4. 10.4 Diffie-Hellman Key Exchange 219

    5. 10.5 The ElGamal Public Key Cryptosystem 221

    6. 10.6 Exercises 223

    7. 10.7 Computer Problems 225

11. 11 Hash Functions 226

    1. 11.1 Hash Functions 226

    2. 11.2 Simple Hash Examples 230

    3. 11.3 The Merkle-Damgård Construction 231

4. 11.4 SHA-2 233

5. 11.5 SHA-3/Keccak 237

6. 11.6 Exercises 242

12. 12 Hash Functions: Attacks and Applications 246

1. 12.1 Birthday Attacks 246

2. 12.2 Multicollisions 249

3. 12.3 The Random Oracle Model 251

4. 12.4 Using Hash Functions to Encrypt 253

5. 12.5 Message Authentication Codes 255

6. 12.6 Password Protocols 256

7. 12.7 Blockchains 262

8. 12.8 Exercises 264

9. 12.9 Computer Problems 268

13. 13 Digital Signatures 269

1. 13.1 RSA Signatures 270

2. 13.2 The ElGamal Signature Scheme 271
3. 13.3 Hashing and Signing 273
4. 13.4 Birthday Attacks on Signatures 274
5. 13.5 The Digital Signature Algorithm 275
6. 13.6 Exercises 276
7. 13.7 Computer Problems 281

14. 14 What Can Go Wrong 282
    1. 14.1 An Enigma “Feature” 282
    2. 14.2 Choosing Primes for RSA 283
    3. 14.3 WEP 284
    4. 14.4 Exercises 288

15. 15 Security Protocols 290
    1. 15.1 Intruders-in-the-Middle and Impostors 290
    2. 15.2 Key Distribution 293
    3. 15.3 Kerberos 299

4. 15.4 Public Key Infrastructures (PKI) 303

5. 15.5 X.509 Certificates 304

6. 15.6 Pretty Good Privacy 309

7. 15.7 SSL and TLS 312

8. 15.8 Secure Electronic Transaction 314

9. 15.9 Exercises 316

16. 16 Digital Cash 318

1. 16.1 Setting the Stage for Digital Economies 319

2. 16.2 A Digital Cash System 320

3. 16.3 Bitcoin Overview 326

4. 16.4 Cryptocurrencies 329

5. 16.5 Exercises 338

17. 17 Secret Sharing Schemes 340

1. 17.1 Secret Splitting 340

2. 17.2 Threshold Schemes 341

3. 17.3 Exercises 346

4. 17.4 Computer Problems 348

18. 18 Games 349

1. 18.1 Flipping Coins over the Telephone 349

2. 18.2 Poker over the Telephone 351

3. 18.3 Exercises 355

19. 19 Zero-Knowledge Techniques 357

1. 19.1 The Basic Setup 357

2. 19.2 The Feige-Fiat-Shamir Identification Scheme 359

3. 19.3 Exercises 361

20. 20 Information Theory 365

1. 20.1 Probability Review 365

2. 20.2 Entropy 367

3. 20.3 Huffman Codes 371

4. 20.4 Perfect Secrecy 373

5. 20.5 The Entropy of English 376

6. 20.6 Exercises 380

21. 21 Elliptic Curves 384

1. 21.1 The Addition Law 384

2. 21.2 Elliptic Curves Mod p 389

3. 21.3 Factoring with Elliptic Curves 393

4. 21.4 Elliptic Curves in Characteristic 2 396

5. 21.5 Elliptic Curve Cryptosystems 399

6. 21.6 Exercises 402

7. 21.7 Computer Problems 407

22. 22 Pairing-Based Cryptography 409

1. 22.1 Bilinear Pairings 409

2. 22.2 The MOV Attack 410

3. 22.3 Tripartite Diffie-Hellman 411

4. 22.4 Identity-Based Encryption 412

5. 22.5 Signatures 414

6. 22.6 Keyword Search 417

7. 22.7 Exercises 419

23. 23 Lattice Methods 421

1. 23.1 Lattices 421

2. 23.2 Lattice Reduction 422

3. 23.3 An Attack on RSA 426

4. 23.4 NTRU 429

5. 23.5 Another Lattice-Based Cryptosystem 433

6. 23.6 Post-Quantum Cryptography? 435

7. 23.7 Exercises 435

24. 24 Error Correcting Codes 437

1. 24.1 Introduction 437

2. 24.2 Error Correcting Codes 442

3. 24.3 Bounds on General Codes 446

4. 24.4 Linear Codes 451

5. 24.5 Hamming Codes 457

6. 24.6 Golay Codes 459

7. 24.7 Cyclic Codes 466

8. 24.8 BCH Codes 472

9. 24.9 Reed-Solomon Codes 479

10. 24.10 The McEliece Cryptosystem 480

11. 24.11 Other Topics 483

12. 24.12 Exercises 483

13. 24.13 Computer Problems 487

25. 25 Quantum Techniques in Cryptography 488

1. 25.1 A Quantum Experiment 488

2. 25.2 Quantum Key Distribution 491

3. 25.3 Shor's Algorithm 493

4. 25.4 Exercises 502

1. A Mathematica® Examples 503
    1. A.1 Getting Started with Mathematica 503
    2. A.2 Some Commands 504
    3. A.3 Examples for Chapter 2 505
    4. A.4 Examples for Chapter 3 508
    5. A.5 Examples for Chapter 5 511
    6. A.6 Examples for Chapter 6 513
    7. A.7 Examples for Chapter 9 514
    8. A.8 Examples for Chapter 10 520
    9. A.9 Examples for Chapter 12 521
    10. A.10 Examples for Chapter 17 521
    11. A.11 Examples for Chapter 18 522
    12. A.12 Examples for Chapter 21 523
2. B Maple® Examples 527
    1. B.1 Getting Started with Maple 527

2. B.2 Some Commands 528

3. B.3 Examples for Chapter 2 529

4. B.4 Examples for Chapter 3 533

5. B.5 Examples for Chapter 5 536

6. B.6 Examples for Chapter 6 538

7. B.7 Examples for Chapter 9 539

8. B.8 Examples for Chapter 10 546

9. B.9 Examples for Chapter 12 547

10. B.10 Examples for Chapter 17 548

11. B.11 Examples for Chapter 18 549

12. B.12 Examples for Chapter 21 551

3. C MATLAB® Examples 555

1. C.1 Getting Started with MATLAB 556

2. C.2 Examples for Chapter 2 560

3. C.3 Examples for Chapter 3 566

4. C.4 Examples for Chapter 5 569

5. C.5 Examples for Chapter 6 571

6. C.6 Examples for Chapter 9 573

7. C.7 Examples for Chapter 10 581

8. C.8 Examples for Chapter 12 581

9. C.9 Examples for Chapter 17 582

10. C.10 Examples for Chapter 18 582

11. C.11 Examples for Chapter 21 585

4. D Sage Examples 591

1. D.1 Computations for Chapter 2 591

2. D.2 Computations for Chapter 3 594

3. D.3 Computations for Chapter 5 595

4. D.4 Computations for Chapter 6 596

5. D.5 Computations for Chapter 9 596

6. D.6 Computations for Chapter 10 597

7. D.7 Computations for Chapter 12 598

8. D.8 Computations for Chapter 17 598

9. D.9 Computations for Chapter 18 598

10. D.10 Computations for Chapter 21 599

5. E Answers and Hints for Selected Odd-Numbered Exercises 601

6. F Suggestions for Further Reading 607

7. Bibliography 608

8. Index 615

Preface

This book is based on a course in cryptography at the upper-level undergraduate and beginning graduate level that has been given at the University of Maryland since 1997, and a course that has been taught at Rutgers University since 2003. When designing the courses, we decided on the following requirements:

- The courses should be up-to-date and cover a broad selection of topics from a mathematical point of view.
- The material should be accessible to mathematically mature students having little background in number theory and computer programming.

- There should be examples involving numbers large enough to demonstrate how the algorithms really work.

We wanted to avoid concentrating solely on RSA and discrete logarithms, which would have made the courses mostly about number theory. We also did not want to focus on protocols and how to hack into friends' computers. That would have made the courses less mathematical than desired.

There are numerous topics in cryptology that can be discussed in an introductory course. We have tried to include many of them. The chapters represent, for the most part, topics that were covered during the different semesters we taught the course. There is certainly more material here than could be treated in

most one-semester courses. The first thirteen chapters represent the core of the material. The choice of which of the remaining chapters are used depends on the level of the students and the objectives of the lecturer.

The chapters are numbered, thus giving them an ordering. However, except for Chapter 3 on number theory, which pervades the subject, the chapters are fairly independent of each other and can be covered in almost any reasonable order. Since students have varied backgrounds in number theory, we have collected the basic number theory facts together in Chapter 3 for ease of reference; however, we recommend introducing these concepts gradually

throughout the course as they are needed.

The chapters on information theory, elliptic curves, quantum cryptography, lattice methods, and error correcting codes are somewhat more mathematical than the others. The chapter on error correcting codes was included, at the suggestion of several reviewers, because courses that include introductions to both cryptology and coding theory are fairly common.

Computer Examples

Suppose you want to give an example for RSA. You could choose two one-digit primes and pretend to be working with fifty-digit primes, or you could use

your favorite software package to do an actual example with large primes. Or perhaps you are working with shift ciphers and are trying to decrypt a message by trying all 26 shifts of the ciphertext. This should also be done on a computer.

Additionally, at the end of the book are appendices containing computer examples written in each of Mathematica®, Maple®, MATLAB®, and Sage that show how to do such calculations. These languages were chosen because they are user friendly and do not require prior programming experience. Although the course has been taught successfully without computers, these examples are an

integral part of the book and should be studied, if at all possible. Not only do they contain numerical examples of how to do certain computations but also they demonstrate important ideas and issues that arise. They were placed at the end of the book because of the logistic and aesthetic problems of including extensive computer examples in these languages at the ends of chapters.

Additionally, programs available in Mathematica, Maple, and MATLAB can be downloaded from the Web site (`bit.ly/2JbcS6p`). Homework problems (the computer problems in various chapters) based on the software allow students to play with examples individually. Of course, students having

more programming background could write their own programs instead. In a classroom, all that is needed is a computer (with one of the languages installed) and a projector in order to produce meaningful examples as the lecture is being given.

New to the Third Edition

Two major changes have informed this edition: Changes to the field of cryptography and a change in the format of the text. We address these issues separately, although there is an interplay between the two:

Content Changes

Cryptography is a quickly changing

field. We have made many changes to the text since the last edition:

- Reorganized content previously in two chapters to four separate chapters on Stream Ciphers (including RC4), Block Ciphers, DES and AES (Chapters 5–8, respectively). The RC4 material, in particular, is new.

- Heavily revised the chapters on hash functions. Chapter 11 (Hash functions) now includes sections on SHA-2 and SHA-3. Chapter 12 (Hash functions: Attacks and Applications) now includes material on message authentication codes, password protocols, and blockchains.

- The short section on the one-time pad has been expanded to become Chapter 4, which includes sections on multiple use of the one-time pad, perfect secrecy, and ciphertext indistinguishability.

- Added Chapter 14, "What Can Go Wrong," which shows what can happen when cryptographic algorithms are used or designed incorrectly.

- Expanded Chapter 16 on digital cash to include Bitcoin and cryptocurrencies.

- Added Chapter 22, which gives an introduction to

Pairing-Based Cryptography.

- Updated the exposition throughout the book to reflect recent developments.
- Added references to the Maple, Mathematica, MATLAB, and Sage appendices in relevant locations in the text.
- Added many new exercises.
- Added a section at the back of the book that contains answers or hints to a majority of the odd-numbered problems.

Format Changes

A focus of this revision was transforming the text from a print-based learning tool to a digital learning tool. The eText is therefore filled with content and tools that will help bring the content of the course to life for students in new ways and help improve instruction. Specifically, the following

are features that are available only in the eText:

- Interactive Examples. We have added a number of opportunities for students to interact with content in a dynamic manner in order to build or enhance understanding. Interactive examples allow students to explore concepts in ways that are not possible without technology.

- Quick Questions. These questions, built into the narrative, provide opportunities for students to check and clarify understanding. Some help address potential misconceptions.

- Notes, Labels, and Highlights. Notes can be added to the eText by instructors. These notes are visible to all students in the course, allowing instructors to add their personal observations or directions to important topics, call out need-to-know information, or clarify difficult concepts. Students can add their own notes, labels, and highlights to the eText, helping them focus on what they need to study. The customizable Notebook allows students to filter, arrange, and group their notes in a way that makes sense to them.

- Dashboard. Instructors can create reading

assignments and see the time spent in the eText so that they can plan more effective instruction.

- Portability. Portable access lets students read their eText whenever they have a moment in their day, on Android and iOS mobile phones and tablets. Even without an Internet connection, offline reading ensures students never miss a chance to learn.

- Ease-of-Use. Straightforward setup makes it easy for instructors to get their class up and reading quickly on the first day of class. In addition, Learning Management System (LMS) integration provides institutions, instructors, and students with single sign-on access to the eText via many popular LMSs.

- Supplements. An Instructors' Solutions Manual can be downloaded by qualified instructors from the textbook's webpage at `www.pearson.com`.

Acknowledgments

Many people helped and provided encouragement during the preparation of this book. First, we would like to thank

our students, whose enthusiasm, insights, and suggestions contributed greatly. We are especially grateful to many people who have provided corrections and other input, especially Bill Gasarch, Jeff Adams, Jonathan Rosenberg, and Tim Strobell. We would like to thank Wenyuan Xu, Qing Li, and Pandurang Kamat, who drew several of the diagrams and provided feedback on the new material for the second edition. We have enjoyed working with the staff at Pearson, especially Jeff Weidenaar and Tara Corpuz.

The reviewers deserve special thanks: their suggestions on the exposition and the organization of the topics greatly enhanced the final result. The reviewers

marked with an asterisk (*) provided input for this edition.

- * Anurag Agarwal, Rochester Institute of Technology
- * Pradeep Atrey, University at Albany

  Eric Bach, University of Wisconsin

  James W. Brewer, Florida Atlantic University

  Thomas P. Cahill, NYU

  Agnes Chan, Northeastern University
- * Nathan Chenette, Rose-Hulman Institute of Technology
- * Claude Crépeau, McGill University
- * Reza Curtmola, New Jersey Institute of Technology
- * Ahmed Desoky, University of Louisville

  Anthony Ephremides, University of Maryland, College Park
- * David J. Fawcett, Lawrence Tech University

- * Jason Gibson, Eastern Kentucky University
- * K. Gopalakrishnan, East Carolina University

  David Grant, University of Colorado, Boulder

  Jugal K. Kalita, University of Colorado, Colorado Springs
- * Saroja Kanchi, Kettering University
- * Andrew Klapper, University of Kentucky
- * Amanda Knecht, Villanova University

  Edmund Lamagna, University of Rhode Island
- * Aihua Li, Montclair State University
- * Spyros S. Magliveras, Florida Atlantic University
- * Nathan McNew, Towson University
- * Nick Novotny, IUPUI

  David M. Pozar, University of Massachusetts, Amherst
- * Emma Previato, Boston University
- * Hamzeh Roumani, York University

- * Bonnie Saunders, University of Illinois, Chicago
- * Ravi Shankar, University of Oklahoma
- * Ernie Stitzinger, North Carolina State
- * Armin Straub, University of South Alabama

  J. Felipe Voloch, University of Texas, Austin

  Daniel F. Warren, Naval Postgraduate School
- * Simon Whitehouse, Alfred State College

  Siman Wong, University of Massachusetts, Amherst
- * Huapeng Wu, University of Windsor

Wade thanks Nisha Gilra, who provided encouragement and advice; Sheilagh O'Hare for introducing him to the field of cryptography; and K. J. Ray Liu for his support. Larry thanks Susan Zengerle and Patrick Washington for their patience, help, and encouragement

during the writing of this book.

Of course, we welcome suggestions and corrections. An errata page can be found at (bit.ly/2J8nN0w) or at the link on the book's general Web site (bit.ly/2T544yu).

Wade Trappe

trappe@winlab.rutgers.edu

Lawrence C. Washington

lcw@math.umd.edu

Chapter 1 Overview of Cryptography and Its Applications

People have always had a fascination with keeping information away from others. As children, many of us had magic decoder rings for exchanging coded messages with our friends and possibly keeping secrets from parents, siblings, or teachers. History is filled with examples where people tried to keep information secret from adversaries. Kings and generals communicated with their troops using basic cryptographic methods to prevent the enemy from learning sensitive military information. In fact, Julius

Caesar reportedly used a simple cipher, which has been named after him.

As society has evolved, the need for more sophisticated methods of protecting data has increased. Now, with the information era at hand, the need is more pronounced than ever. As the world becomes more connected, the demand for information and electronic services is growing, and with the increased demand comes increased dependency on electronic systems. Already the exchange of sensitive information, such as credit card numbers, over the Internet is common practice. Protecting data and electronic systems is crucial to our way of living.

The techniques needed to protect data

belong to the field of cryptography. Actually, the subject has three names, **cryptography**, **cryptology**, and **cryptanalysis**, which are often used interchangeably. Technically, however, cryptology is the all-inclusive term for the study of communication over nonsecure channels, and related problems. The process of designing systems to do this is called cryptography. Cryptanalysis deals with breaking such systems. Of course, it is essentially impossible to do either cryptography or cryptanalysis without having a good understanding of the methods of both areas.

Often the term **coding theory** is used to describe cryptography; however, this

can lead to confusion. Coding theory deals with representing input information symbols by output symbols called code symbols. There are three basic applications that coding theory covers: compression, secrecy, and error correction. Over the past few decades, the term coding theory has become associated predominantly with error correcting codes. Coding theory thus studies communication over noisy channels and how to ensure that the message received is the correct message, as opposed to cryptography, which protects communication over nonsecure channels.

Although error correcting codes are only a secondary focus of this book, we

should emphasize that, in any real-world system, error correcting codes are used in conjunction with encryption, since the change of a single bit is enough to destroy the message completely in a well-designed cryptosystem.

Modern cryptography is a field that draws heavily upon mathematics, computer science, and cleverness. This book provides an introduction to the mathematics and protocols needed to make data transmission and electronic systems secure, along with techniques such as electronic signatures and secret sharing.

1.1 Secure Communications

In the basic communication scenario, depicted in Figure 1.1, there are two parties, we'll call them Alice and Bob, who want to communicate with each other. A third party, Eve, is a potential eavesdropper.

Figure 1.1 The Basic Communication Scenario for Cryptography.

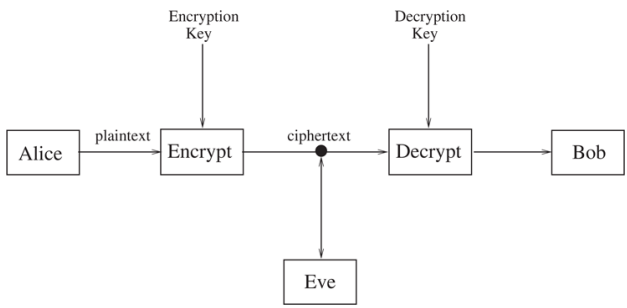


Figure 1.1 Full Alternative Text

When Alice wants to send a message, called the **plaintext**, to Bob, she encrypts it using a method prearranged with Bob. Usually, the encryption method is assumed to be known to Eve; what keeps the message secret is a **key**. When Bob receives the encrypted message, called the **ciphertext**, he changes it back to the plaintext using a decryption key.

Eve could have one of the following goals:

1. Read the message.
2. Find the key and thus read all messages encrypted with that key.
3. Corrupt Alice's message into another message in such a way that Bob will think Alice sent the altered message.

4. Masquerade as Alice, and thus communicate with Bob even though Bob believes he is communicating with Alice.

Which case we're in depends on how evil Eve is. Cases (3) and (4) relate to issues of integrity and authentication, respectively. We'll discuss these shortly. A more active and malicious adversary, corresponding to cases (3) and (4), is sometimes called Mallory in the literature. More passive observers (as in cases (1) and (2)) are sometimes named Oscar. We'll generally use only Eve, and assume she is as bad as the situation allows.

1.1.1 Possible Attacks

There are four main types of attack that Eve might be able to use. The

differences among these types of attacks are the amounts of information Eve has available to her when trying to determine the key. The four attacks are as follows:

1. Ciphertext only: Eve has only a copy of the ciphertext.

2. Known plaintext: Eve has a copy of a ciphertext and the corresponding plaintext. For example, suppose Eve intercepts an encrypted press release, then sees the decrypted release the next day. If she can deduce the decryption key, and if Alice doesn't change the key, Eve can read all future messages. Or, if Alice always starts her messages with "Dear Bob," then Eve has a small piece of ciphertext and corresponding plaintext. For many weak cryptosystems, this suffices to find the key. Even for stronger systems such as the German Enigma machine used in World War II, this amount of information has been useful.

3. Chosen plaintext: Eve gains temporary access to the encryption machine. She cannot open it to find the key; however, she can encrypt a large number of suitably chosen plaintexts and try to use the

resulting ciphertexts to deduce the key.

4. Chosen ciphertext: Eve obtains temporary access to the decryption machine, uses it to “decrypt” several strings of symbols, and tries to use the results to deduce the key.

A chosen plaintext attack could happen as follows. You want to identify an airplane as friend or foe. Send a random message to the plane, which encrypts the message automatically and sends it back. Only a friendly airplane is assumed to have the correct key. Compare the message from the plane with the correctly encrypted message. If they match, the plane is friendly. If not, it’s the enemy. However, the enemy can send a large number of chosen messages to one of your planes and look at the resulting ciphertexts. If this allows them to deduce the key, the enemy can equip

their planes so they can masquerade as friendly.

An example of a known plaintext attack reportedly happened in World War II in the Sahara Desert. An isolated German outpost every day sent an identical message saying that there was nothing new to report, but of course it was encrypted with the key being used that day. So each day the Allies had a plaintext-ciphertext pair that was extremely useful in determining the key. In fact, during the Sahara campaign, General Montgomery was carefully directed around the outpost so that the transmissions would not be stopped.

One of the most important assumptions in modern cryptography is **Kerckhoffs's**

principle: In assessing the security of a cryptosystem, one should always assume the enemy knows the method being used. This principle was enunciated by Auguste Kerckhoffs in 1883 in his classic treatise *La Cryptographie Militaire*. The enemy can obtain this information in many ways. For example, encryption/decryption machines can be captured and analyzed. Or people can defect or be captured. The security of the system should therefore be based on the key and not on the obscurity of the algorithm used. Consequently, we always assume that Eve has knowledge of the algorithm that is used to perform encryption.

1.1.2 Symmetric and Public

Key Algorithms

Encryption/decryption methods fall into two categories: **symmetric key** and **public key**. In symmetric key algorithms, the encryption and decryption keys are known to both Alice and Bob. For example, the encryption key is shared and the decryption key is easily calculated from it. In many cases, the encryption key and the decryption key are the same. All of the classical (pre-1970) cryptosystems are symmetric, as are the more recent Data Encryption Standard (DES) and Advanced Encryption Standard (AES).

Public key algorithms were introduced in the 1970s and revolutionized cryptography. Suppose Alice wants to

communicate securely with Bob, but they are hundreds of kilometers apart and have not agreed on a key to use. It seems almost impossible for them to do this without first getting together to agree on a key, or using a trusted courier to carry the key from one to the other. Certainly Alice cannot send a message over open channels to tell Bob the key, and then send the ciphertext encrypted with this key. The amazing fact is that this problem has a solution, called public key cryptography. The encryption key is made public, but it is computationally infeasible to find the decryption key without information known only to Bob. The most popular implementation is RSA (see Chapter 9), which is based on the difficulty of

factoring large integers. Other versions (see Chapters 10, 23, and 24) are the ElGamal system (based on the discrete log problem), NTRU (lattice based) and the McEliece system (based on error correcting codes).

Here is a nonmathematical way to do public key communication. Bob sends Alice a box and an unlocked padlock. Alice puts her message in the box, locks Bob's lock on it, and sends the box back to Bob. Of course, only Bob can open the box and read the message. The public key methods mentioned previously are mathematical realizations of this idea. Clearly there are questions of authentication that must be dealt with. For example, Eve could intercept

the first transmission and substitute her own lock. If she then intercepts the locked box when Alice sends it back to Bob, Eve can unlock her lock and read Alice's message. This is a general problem that must be addressed with any such system.

Public key cryptography represents what is possibly the final step in an interesting historical progression. In the earliest years of cryptography, security depended on keeping the encryption method secret. Later, the method was assumed known, and the security depended on keeping the (symmetric) key private or unknown to adversaries. In public key cryptography, the method and the encryption key are made public,

and everyone knows what must be done to find the decryption key. The security rests on the fact (or hope) that this is computationally infeasible. It's rather paradoxical that an increase in the power of cryptographic algorithms over the years has corresponded to an increase in the amount of information given to an adversary about such algorithms.

Public key methods are very powerful, and it might seem that they make the use of symmetric key cryptography obsolete. However, this added flexibility is not free and comes at a computational cost. The amount of computation needed in public key algorithms is typically several orders of magnitude more than

the amount of computation needed in algorithms such as DES or AES/ Rijndael. The rule of thumb is that public key methods should not be used for encrypting large quantities of data. For this reason, public key methods are used in applications where only small amounts of data must be processed (for example, digital signatures and sending keys to be used in symmetric key algorithms).

Within symmetric key cryptography, there are two types of ciphers: stream ciphers and block ciphers. In stream ciphers, the data are fed into the algorithm in small pieces (bits or characters), and the output is produced in corresponding small pieces. We

discuss stream ciphers in Chapter 5. In block ciphers, however, a block of input bits is collected and fed into the algorithm all at once, and the output is a block of bits. Mostly we shall be concerned with block ciphers. In particular, we cover two very significant examples. The first is DES, and the second is AES, which was selected in the year 2000 by the National Institute for Standards and Technology as the replacement for DES. Public key methods such as RSA can also be regarded as block ciphers.

Finally, we mention a historical distinction between different types of encryption, namely **codes** and **ciphers**. In a code, words or certain letter

combinations are replaced by codewords (which may be strings of symbols). For example, the British navy in World War I used 03680C, 36276C, and 50302C to represent *shipped at, shipped by*, and *shipped from*, respectively. Codes have the disadvantage that unanticipated words cannot be used. A cipher, on the other hand, does not use the linguistic structure of the message but rather encrypts every string of characters, meaningful or not, by some algorithm. A cipher is therefore more versatile than a code. In the early days of cryptography, codes were commonly used, sometimes in conjunction with ciphers. They are still used today; covert operations are often given code names. However, any secret that is to remain

secure needs to be encrypted with a cipher. In this book, we'll deal exclusively with ciphers.

1.1.3 Key Length

The security of cryptographic algorithms is a difficult property to measure. Most algorithms employ keys, and the security of the algorithm is related to how difficult it is for an adversary to determine the key. The most obvious approach is to try every possible key and see which ones yield meaningful decryptions. Such an attack is called a **brute force attack**. In a brute force attack, the length of the key is directly related to how long it will take to search the entire keyspace. For example, if a key is 16 bits long, then

there are 216=65536 possible keys. The DES algorithm has a 56-bit key and thus has 256 ≈ 7.2×1016 possible keys.

In many situations we'll encounter in this book, it will seem that a system can be broken by simply trying all possible keys. However, this is often easier said than done. Suppose you need to try 1030 possibilities and you have a computer that can do 109 such calculations each second. There are around 3×107 seconds in a year, so it would take a little more than 3×1013 years to complete the task, longer than the predicted life of the universe.

Longer keys are advantageous but are not guaranteed to make an adversary's task difficult. The algorithm itself also

plays a critical role. Some algorithms might be able to be attacked by means other than brute force, and some algorithms just don't make very efficient use of their keys' bits. This is a very important point to keep in mind. Not all 128-bit algorithms are created equal!

For example, one of the easiest cryptosystems to break is the substitution cipher, which we discuss in Section 2.4. The number of possible keys is 26! ≈ 4×1026. In contrast, DES (see Chapter 7) has only 256 ≈ 7.2×1016 keys. But it typically takes over a day on a specially designed computer to find a DES key. The difference is that an attack on a substitution cipher uses the underlying

structure of the language, while the attack on DES is by brute force, trying all possible keys.

A brute force attack should be the last resort. A cryptanalyst always hopes to find an attack that is faster. Examples we'll meet are frequency analysis (for the substitution and Vigenère ciphers) and birthday attacks (for discrete logs).

We also warn the reader that just because an algorithm seems secure now, we can't assume that it will remain so. Human ingenuity has led to creative attacks on cryptographic protocols. There are many examples in modern cryptography where an algorithm or protocol was successfully attacked because of a loophole presented by poor

implementation, or just because of advances in technology. The DES algorithm, which withstood 20 years of cryptographic scrutiny, ultimately succumbed to attacks by a well-designed parallel computer. Even as you read this book, research in quantum computing is underway, which could dramatically alter the terrain of future cryptographic algorithms.

For example, the security of several systems we'll study depends on the difficulty of factoring large integers, say of around 600 digits. Suppose you want to factor a number n of this size. The method used in elementary school is to divide n by all of the primes up to the square root of n. There are

approximately 1.4×10297 primes less than 10300. Trying each one is impossible. The number of electrons in the universe is estimated to be less than 1090. Long before you finish your calculation, you'll get a call from the electric company asking you to stop. Clearly, more sophisticated factoring algorithms must be used, rather than this brute force type of attack. When RSA was invented, there were some good factoring algorithms available, but it was predicted that a 129-digit number such as the RSA challenge number (see Chapter 9) would not be factored within the foreseeable future. However, advances in algorithms and computer architecture have made such factorizations fairly routine (although

they still require substantial computing resources), so now numbers of several hundred digits are recommended for security. But if a full-scale quantum computer is ever built, factorizations of even these numbers will be easy, and the whole RSA scheme (along with many other methods) will need to be reconsidered.

A natural question, therefore, is whether there are any unbreakable cryptosystems, and, if so, why aren't they used all the time?

The answer is yes; there is a system, known as the one-time pad, that is unbreakable. Even a brute force attack will not yield the key. But the unfortunate truth is that the expense of

using a one-time pad is enormous. It requires exchanging a key that is as long as the plaintext, and even then the key can only be used once. Therefore, one opts for algorithms that, when implemented correctly with the appropriate key size, are unbreakable in any reasonable amount of time.

An important point when considering key size is that, in many cases, one can mathematically increase security by a slight increase in key size, but this is not always practical. If you are working with chips that can handle words of 64 bits, then an increase in the key size from 64 to 65 bits could mean redesigning your hardware, which could be expensive. Therefore, designing good

cryptosystems involves both mathematical and engineering considerations.

Finally, we need a few words about the size of numbers. Your intuition might say that working with a 20-digit number takes twice as long as working with a 10-digit number. That is true in some algorithms. However, if you count up to 1010, you are not even close to 1020; you are only one 10 billionth of the way there. Similarly, a brute force attack against a 60-bit key takes a billion times longer than one against a 30-bit key.

There are two ways to measure the size of numbers: the actual magnitude of the number n, and the number of digits in its decimal representation (we could also

use its binary representation), which is approximately log10(n). The number of single-digit multiplications needed to square a k-digit number n, using the standard algorithm from elementary school, is k2, or approximately (log 10 n) 2 . The number of divisions needed to factor a number n by dividing by all primes up to the square root of n is around n1/2. An algorithm that runs in time a power of log n is much more desirable than one that runs in time a power of n. In the present example, if we double the number of digits in n, the time it takes to square n increases by a factor of 4, while the time it takes to factor n increases enormously. Of course, there are better algorithms available for both of these operations,

but, at present, factorization takes significantly longer than multiplication.

We'll meet algorithms that take time a power of logn to perform certain calculations (for example, finding greatest common divisors and doing modular exponentiation). There are other computations for which the best known algorithms run only slightly better than a power of n (for example, factoring and finding discrete logarithms). The interplay between the fast algorithms and the slower ones is the basis of several cryptographic algorithms that we'll encounter in this book.

1.2 Cryptographic Applications

Cryptography is not only about encrypting and decrypting messages, it is also about solving real-world problems that require information security. There are four main objectives that arise:

1. Confidentiality: Eve should not be able to read Alice's message to Bob. The main tools are encryption and decryption algorithms.

2. Data integrity: Bob wants to be sure that Alice's message has not been altered. For example, transmission errors might occur. Also, an adversary might intercept the transmission and alter it before it reaches the intended recipient. Many cryptographic primitives, such as hash functions, provide methods to detect data manipulation by malicious or accidental

adversaries.

3. Authentication: Bob wants to be sure that only Alice could have sent the message he received. Under this heading, we also include identification schemes and password protocols (in which case, Bob is the computer). There are actually two types of authentication that arise in cryptography: entity authentication and data-origin authentication. Often the term *identification* is used to specify entity authentication, which is concerned with proving the identity of the parties involved in a communication. Data-origin authentication focuses on tying the information about the origin of the data, such as the creator and time of creation, with the data.

4. Non-repudiation: Alice cannot claim she did not send the message. Non-repudiation is particularly important in electronic commerce applications, where it is important that a consumer cannot deny the authorization of a purchase.

Authentication and non-repudiation are closely related concepts, but there is a difference. In a symmetric key cryptosystem, Bob can be sure that a

message comes from Alice (or someone who knows Alice's key) since no one else could have encrypted the message that Bob decrypts successfully. Therefore, authentication is automatic. However, he cannot prove to anyone else that Alice sent the message, since he could have sent the message himself. Therefore, non-repudiation is essentially impossible. In a public key cryptosystem, both authentication and non-repudiation can be achieved (see Chapters 9, 13, and 15).

Much of this book will present specific cryptographic applications, both in the text and as exercises. Here is an overview.

Digital signatures: One of the most

important features of a paper and ink letter is the signature. When a document is signed, an individual's identity is tied to the message. The assumption is that it is difficult for another person to forge the signature onto another document. Electronic messages, however, are very easy to copy exactly. How do we prevent an adversary from cutting the signature off one document and attaching it to another electronic document? We shall study cryptographic protocols that allow for electronic messages to be signed in such a way that everyone believes that the signer was the person who signed the document, and such that the signer cannot deny signing the document.

Identification: When logging into a machine or initiating a communication link, a user needs to identify herself or himself. But simply typing in a user name is not sufficient as it does not prove that the user is really who he or she claims to be. Typically a password is used. We shall touch upon various methods for identifying oneself. In the chapter on DES we discuss password files. Later, we present the Feige-Fiat-Shamir identification scheme, which is a zero-knowledge method for proving identity without revealing a password.

Key establishment: When large quantities of data need to be encrypted, it is best to use symmetric key encryption algorithms. But how does

Alice give the secret key to Bob when she doesn't have the opportunity to meet him personally? There are various ways to do this. One way uses public key cryptography. Another method is the Diffie-Hellman key exchange algorithm. A different approach to this problem is to have a trusted third party give keys to Alice and Bob. Two examples are Blom's key generation scheme and Kerberos, which is a very popular symmetric cryptographic protocol that provides authentication and security in key exchange between users on a network.

Secret sharing: In Chapter 17, we introduce secret sharing schemes. Suppose that you have a combination to

a bank safe, but you don't want to trust any single person with the combination to the safe. Rather, you would like to divide the combination among a group of people, so that at least two of these people must be present in order to open the safe. Secret sharing solves this problem.

Security protocols: How can we carry out secure transactions over open channels such as the Internet, and how can we protect credit card information from fraudulent merchants? We discuss various protocols, such as SSL and SET.

Electronic cash: Credit cards and similar devices are convenient but do not provide anonymity. Clearly a form of electronic cash could be useful, at least

to some people. However, electronic entities can be copied. We give an example of an electronic cash system that provides anonymity but catches counterfeiters, and we discuss cryptocurrencies, especially Bitcoin.

Games: How can you flip coins or play poker with people who are not in the same room as you? Dealing the cards, for example, presents a problem. We show how cryptographic ideas can solve these problems.

Chapter 2 Classical Cryptosystems

Methods of making messages unintelligible to adversaries have been important throughout history. In this chapter we shall cover some of the older cryptosystems that were primarily used before the advent of the computer. These cryptosystems are too weak to be of much use today, especially with computers at our disposal, but they give good illustrations of several of the important ideas of cryptology.

First, for these simple cryptosystems, we make some conventions.

- *plaintext* will be written in lowercase letters and *CIPHERTEXT* will be written in capital letters (except in the computer problems).

- The letters of the alphabet are assigned numbers as follows:

  abcdefghijklmnop012345678910111213141 5qrstuvwxyz161718192

  Note that we start with a=0, so z is letter number 25. Because many people are accustomed to a being 1 and z being 26, the present convention can be annoying, but it is standard for the elementary cryptosystems that we'll consider.

- Spaces and punctuation are omitted. This is even more annoying, but it is almost always possible to replace the spaces in the plaintext after decrypting. If spaces were left in, there would be two choices. They could be left as spaces; but this yields so much information on the structure of the message that decryption becomes easier. Or they could be encrypted; but then they would dominate frequency counts (unless the message averages at least eight letters per word), again simplifying decryption.

Note: In this chapter, we'll be using some concepts from number theory,

especially modular arithmetic. If you are not familiar with congruences, you should read the first three sections of Chapter 3 before proceeding.

2.1 Shift Ciphers

One of the earliest cryptosystems is often attributed to Julius Caesar. Suppose he wanted to send a plaintext such as

gaul is divided into three parts

but he didn't want Brutus to read it. He shifted each letter backwards by three places, so d became A, e became B, f became C, etc. The beginning of the alphabet wrapped around to the end, so a became X, b became Y, and c became Z. The ciphertext was then

DXRIFPAFSFABAFKQLQEOBBMXOQP.

Decryption was accomplished by

shifting FORWARD by three spaces (and trying to figure out how to put the spaces back in).

We now give the general situation. *If you are not familiar with modular arithmetic, read the first few pages of Chapter 3 before continuing.*

Label the letters as integers from 0 to 25. The key is an integer κ with $0 \le \kappa \le 25$. The encryption process is

$$x \mapsto x+\kappa \pmod{26}.$$

Decryption is $x \mapsto x-\kappa \pmod{26}$. For example, Caesar used $\kappa=23 \equiv -3$.

Let's see how the four types of attack work.

1. Ciphertext only: Eve has only the ciphertext. Her best strategy is an exhaustive search, since there are only 26 possible keys. See Example 1 in the Computer Appendices. If the message is longer than a few letters (we will make this more precise later when we discuss entropy), it is unlikely that there is more than one meaningful message that could be the plaintext. If you don't believe this, try to find some words of four or more letters that are shifts of each other. Three such words are given in Exercises 1 and 2. Another possible attack, if the message is sufficiently long, is to do a frequency count for the various letters. The letter e occurs most frequently in most English texts. Suppose the letter L appears most frequently in the ciphertext. Since e=4 and L=11, a reasonable guess is that $\kappa = 11 - 4 = 7$. However, for shift ciphers this method takes much longer than an exhaustive search, plus it requires many more letters in the message in order for it to work (anything short, such as this, might not contain a common symbol, thus changing statistical counts).

2. Known plaintext: If you know just one letter of the plaintext along with the corresponding letter of ciphertext, you can deduce the key. For example, if you know t(=19) encrypts to D(=3), then the key is $\kappa \equiv 3 - 19 \equiv -16 \equiv 10 \pmod{26}$.

3. Chosen plaintext: Choose the letter a as the

plaintext. The ciphertext gives the key. For example, if the ciphertext is H, then the key is 7.

4. Chosen ciphertext: Choose the letter A as ciphertext. The plaintext is the negative of the key. For example, if the plaintext is h, the key is $-7 \equiv 19 \pmod{26}$.

2.2 Affine Ciphers

The shift ciphers may be generalized and slightly strengthened as follows. Choose two integers α and β, with $\gcd(\alpha, 26)=1$, and consider the function (called an *affine function*)

$$x \mapsto \alpha x+\beta \pmod{26}.$$

For example, let $\alpha=9$ and $\beta=2$, so we are working with $9x+2$. Take a plaintext letter such as h(=7). It is encrypted to $9\cdot 7+2 \equiv 65 \equiv 13 \pmod{26}$, which is the letter N. Using the same function, we obtain

$$\text{affine} \mapsto \text{CVVWPM}.$$

How do we decrypt? If we were

working with rational numbers rather than mod 26, we would start with $y=9x+2$ and solve: $x=19(y-2)$. But 19 needs to be reinterpreted when we work mod 26. Since $\gcd(9, 26)=1$, there is a multiplicative inverse for 9 (mod 26) (if this last sentence doesn't make sense to you, read Section 3.3 now). In fact, $9\cdot 3 \equiv 1 \pmod{26}$, so 3 is the desired inverse and can be used in place of 19. We therefore have

$$x \equiv 3(y-2) \equiv 3y-6 \equiv 3y+20 \pmod{26}.$$

Let's try this. The letter V(=21) is mapped to $3\cdot 21+20 \equiv 83 \equiv 5 \pmod{26}$, which is the letter f. Similarly, we see that the ciphertext CVVWPM is decrypted back to *affine*. For more examples, see Examples 2 and 3 in the

Suppose we try to use the function 13x +4 as our encryption function. We obtain

input↦ ERRER.

If we alter the input, we obtain

alter↦ ERRER.

Clearly this function leads to errors. It is impossible to decrypt, since several plaintexts yield the same ciphertext. In particular, we note that encryption must be one-to-one, and this fails in the present case.

What goes wrong in this example? If we solve y=13x+4, we obtain x=113(y−4). But 113 does not exist mod 26 since

gcd(13, 26)=13 ≠ 1. More generally, it can be shown that αx+β is a one-to-one function mod 26 if and only if gcd(α, 26)=1. In this case, decryption uses $x \equiv \alpha^* y - \alpha^* \beta \pmod{26}$, where $\alpha\alpha^* \equiv 1 \pmod{26}$. So decryption is also accomplished by an affine function.

The key for this encryption method is the pair (α, β). There are 12 possible choices for α with gcd(α, 26)=1 and there are 26 choices for β (since we are working mod 26, we only need to consider α and β between 0 and 25). Therefore, there are 12·26=312 choices for the key.

Let's look at the possible attacks.

1. Ciphertext only: An exhaustive search through all

312 keys would take longer than the corresponding search in the case of the shift cipher; however, it would be very easy to do on a computer. When all possibilities for the key are tried, a fairly short ciphertext, say around 20 characters, will probably correspond to only one meaningful plaintext, thus allowing the determination of the key. It would also be possible to use frequency counts, though this would require much longer texts.

2. Known plaintext: With a little luck, knowing two letters of the plaintext and the corresponding letters of the ciphertext suffices to find the key. In any case, the number of possibilities for the key is greatly reduced and a few more letters should yield the key.

   For example, suppose the plaintext starts with *if* and the corresponding ciphertext is PQ. In numbers, this means that 8 (=i) maps to 15 (=P) and 5 maps to 16. Therefore, we have the equations

   $$8\alpha+\beta \equiv 15 \text{ and } 5\alpha+\beta \equiv 16 \pmod{26}.$$

   Subtracting yields $3\alpha \equiv -1 \equiv 25 \pmod{26}$, which has the unique solution $\alpha=17$. Using the first equation, we find $8\cdot 17+\beta \equiv 15 \pmod{26}$, which yields $\beta=9$.

Suppose instead that the plaintext *go* corresponds to the ciphertext TH. We obtain the equations

$$6\alpha+\beta \equiv 19 \text{ and } 14\alpha+\beta \equiv 7 \pmod{26}.$$

Subtracting yields $-8\alpha \equiv 12 \pmod{26}$. Since $\gcd(-8, 26)=2$, this has two solutions: $\alpha=5, 18$. The corresponding values of β are both 15 (this is not a coincidence; it will always happen this way when the coefficients of α in the equations are even). So we have two candidates for the key: (5, 15) and (18, 15). However, $\gcd(18, 26) \neq 1$ so the second is ruled out. Therefore, the key is (5, 15).

The preceding procedure works unless the gcd we get is 13 (or 26). In this case, use another letter of the message, if available.

If we know only one letter of plaintext, we still get a relation between α and β. For example, if we only know that g in plaintext corresponds to T in ciphertext, then we have $6\alpha+\beta \equiv 19 \pmod{26}$. There are 12 possibilities for α and each gives one corresponding β. Therefore, an exhaustive search through the 12 keys should yield the correct key.

3. Chosen plaintext: Choose ab as the plaintext. The first character of the ciphertext will be $\alpha \cdot 0+\beta=\beta$, and the second will be $\alpha+\beta$. Therefore, we can find the key.

4. Chosen ciphertext: Choose AB as the ciphertext. This yields the decryption function of the form $x=\alpha 1y+\beta 1$. We could solve for y and obtain the encryption key. But why bother? We have the decryption function, which is what we want.

2.3 The Vigenère Cipher

A variation of the shift cipher was invented back in the sixteenth century. It is often attributed to Vigenère, though Vigenère's encryption methods were more sophisticated. Well into the twentieth century, this cryptosystem was thought by many to be secure, though Babbage and Kasiski had shown how to attack it during the nineteenth century. In the 1920s, Friedman developed additional methods for breaking this and related ciphers.

The key for the encryption is a vector, chosen as follows. First choose a key length, for example, 6. Then choose a

vector of this size whose entries are integers from 0 to 25, for example k=(21, 4, 2, 19, 14, 17). Often the key corresponds to a word that is easily remembered. In our case, the word is *vector*. The security of the system depends on the fact that neither the keyword nor its length is known.

To encrypt the message using the k in our example, we take first the letter of the plaintext and shift by 21. Then shift the second letter by 4, the third by 2, and so on. Once we get to the end of the key, we start back at its first entry, so the seventh letter is shifted by 21, the eighth letter by 4, etc. Here is a diagram of the encryption process.

(plaintext) (key) (ciphertext) hereishowitworks21421914172142191417214219

A known plaintext attack will succeed if enough characters are known since the key is simply obtained by subtracting the plaintext from the ciphertext mod 26. A chosen plaintext attack using the plaintext aaaaa… will yield the key immediately, while a chosen ciphertext attack with AAAAA… yields the negative of the key. But suppose you have only the ciphertext. It was long thought that the method was secure against a ciphertext-only attack. However, it is easy to find the key in this case, too.

The cryptanalysis uses the fact that in most English texts the frequencies of letters are not equal. For example, e occurs much more frequently than x.

These frequencies have been tabulated in [Beker-Piper] and are provided in Table 2.1.

Table 2.1 Frequencies of Letters in English

a	b	c	d	e	f	g	h	i	j
.082	.015	.028	.043	.127	.022	.020	.061	.070	.002
k	l	m	n	o	p	q	r	s	t
.008	.040	.024	.067	.075	.019	.001	.060	.063	.091
u	v	w	x	y	z				
.028	.010	.023	.001	.020	.001				

Table 2.1 Full Alternative Text

Of course, variations can occur, though usually it takes a certain amount of effort to produce them. There is a book Gadsby by Ernest Vincent Wright that does not contain the letter e. Even more impressive is the book La Disparition by George Perec, written in French, which also does not have a single e (not only are there the usual problems with verbs,

etc., but almost all feminine nouns and adjectives must be avoided). There is an English translation by Gilbert Adair, A Void, which also does not contain e. But generally we can assume that the above gives a rough estimate of what usually happens, as long as we have several hundred characters of text.

If we had a simple shift cipher, then the letter e, for example, would always appear as a certain ciphertext letter, which would then have the same frequency as that of e in the original text. Therefore, a frequency analysis would probably reveal the key. However, in the preceding example of a Vigenère cipher, the letter e appears as both I and X. If we had used a longer

plaintext, e would probably have been encrypted as each of Z, I, G, X, S, and V, corresponding to the shifts 21, 4, 2, 19, 14, 17. But the occurrences of Z in a ciphertext might not come only from e. The letter v is also encrypted to Z when its position in the text is such that it is shifted by 4. Similarly, x, g, l, and i can contribute Z to the ciphertext, so the frequency of Z is a combination of that of e, v, x, g, l, and i from the plaintext. Therefore, it appears to be much more difficult to deduce anything from a frequency count. In fact, the frequency counts are usually smoothed out and are much closer to 1/26 for each letter of ciphertext. At least, they should be much closer than the original distribution for English letters.

Here is a more substantial example. This example is also treated in Example 4 in the Computer Appendices. The ciphertext is the following:

VVHQWVVRHMUSGJGTHKIHTSSEJCHLSE
KCUHWAUGLQHNSLRLJSHBLTSPISPRDX
PWQTWVSPGOELKCQYFNSVWLJSNIQKG
KXZGYHCECMEIUJOQKWFWVEFQHKIJR
HLSFQTWLAUQRHWDMWLGUSGIKKFLR
GGVEYGGZMLJCXXLJSVPAIVWIKVRDRY
APUUISFPBTGNWWMUCZRVTWGLRWUC

The frequencies are as follows:

Note that there is no letter whose frequency is significantly larger than the others. As discussed previously, this is because e, for example, gets spread among several letters during the encryption process.

How do we decrypt the message? There are two steps: finding the key length and finding the key. In the following, we'll first show how to find the key length and then give one way to find the key. After an explanation of why the method for finding the key works, we give an

alternative way to find the key.

2.3.1 Finding the Key Length

Write the ciphertext on a long strip of paper, and again on another long strip. Put one strip above the other, but displaced by a certain number of places (the potential key length). For example, for a displacement of two we have the following:

Mark a * each time a letter and the one below it are the same, and count the total number of coincidences. In the text just listed, we have two coincidences so far. If we had continued for the entire ciphertext, we would have counted 14 of them. If we do this for different displacements, we obtain the following data:

We have the most coincidences for a shift of 5. As we explain later, this is the best guess for the length of the key. This method works very quickly, even without a computer, and usually yields the key length.

2.3.2 Finding the Key: First Method

Now suppose we have determined the key length to be 5, as in our example. Look at the 1st, 6th, 11th, ... letters and see which letter occurs most frequently. We obtain

The most frequent is G, though J, K, C are close behind. However, J=e would mean a shift of 5, hence C=x. But this would yield an unusually high frequency for x in the plaintext. Similarly, K=e would mean P=j and Q=k, both of which have too high frequencies. Finally, C=e would require V=x, which is unlikely to be the case.

Therefore, we decide that G=e and the first element of the key is 2=c.

We now look at the 2nd, 7th, 12th, ... letters. We find that G occurs 10 times and S occurs 12 times, and the other letters are far behind. If G=e, then S=q, which should not occur 12 times in the plaintext. Therefore, S=e and the second element of the key is 14=o.

Now look at the 3rd, 8th, 13th, ... letters. The frequencies are

The initial guess that L=e runs into problems; for example, R=k and E=x have too high frequency and A=t has too low. Similarly, V=e and W=e do not seem likely. The best choice is H=e and therefore the third key element is 3=d.

The 4th, 9th, 14th, ... letters yield 4=e as the fourth element of the key. Finally, the 5th, 10th, 15th, ... letters yield 18=s as the final key element. Our guess for the key is therefore

{2, 14, 3, 4, 18}={c, o, d, e, s}.

As we saw in the case of the 3rd, 8th,

13th, ... letters (this also happened in the 5th, 10th, 15th, ... case), if we take every fifth letter we have a much smaller sample of letters on which we are doing a frequency count. Another letter can overtake e in a short sample. But it is probable that most of the high-frequency letters appear with high frequencies, and most of the low-frequency ones appear with low frequencies. As in the present case, this is usually sufficient to identify the corresponding entry in the key.

Once a potential key is found, test it by using it to decrypt. It should be easy to tell whether it is correct.

In our example, the key is conjectured to be (2, 14, 3, 4, 18). If we decrypt the

ciphertext using this key, we obtain

themethodusedforthepreparationandreadingofco

simpleintheextremeandatthesametimeimpossible

nunlessthekeyisknowntheeasewithwhichthekeyn

anotherpointinfavoroftheadoptionofthiscodebyth

gtotransmitimportantmessageswithouttheslighte

heirmessagesbeingreadbypoliticalorbusinessriva

This passage is taken from a short article in Scientific American, Supplement LXXXIII (January 27, 1917), page 61. A short explanation of the Vigenère cipher is given, and the preceding passage expresses an opinion

as to its security.

Before proceeding to a second method for finding the key, we give an explanation of why the procedure given earlier finds the key length. In order to avoid confusion, we note that when we use the word “shift” for a letter, we are referring to what happens during the Vigenère encryption process.

We also will be shifting elements in vectors. However, when we slide one strip of paper to the right or left relative to the other strip, we use the word “displacement.”

Put the frequencies of English letters into a vector:

A0=(.082, .015, .028, … , .020, .001).

Let Ai be the result of shifting A0 by i spaces to the right. For example,

A2=(.020, .001, .082, .015, …).

The dot product of A0 with itself is

A0·A0=(.082)2+(.015)2+⋯=.066.

Of course, Ai·Ai is also equal to .066 since we get the same sum of products, starting with a different term. However, the dot products of Ai·Aj are much lower when i ≠ j, ranging from .031 to .045:

The dot product depends only on $|i-j|$. This can be seen as follows. The entries in the vectors are the same as those in A0, but shifted. In the dot product, the ith entry of A0 is multiplied by the jth

entry, the (i+1)st times the (j+1)st, etc. So each element is multiplied by the element j−i positions removed from it. Therefore, the dot product depends only on the difference i−j. However, by reversing the roles of i and j, and noting that Ai·Aj=Aj·Ai, we see that i−j and j−i give the same dot products, so the dot product only depends on |i−j|. In the preceding table, we only needed to compute up to |i−j|=13. For example, i −j=17 corresponds to a shift by 17 in one direction, or 9 in the other direction, so i−j=9 will give the same dot product.

The reason A0·A0 is higher than the other dot products is that the large numbers in the vectors are paired with large numbers and the small ones are

paired with small. In the other dot products, the large numbers are paired somewhat randomly with other numbers. This lessens their effect. For another reason that A0·A0 is higher than the other dot products, see Exercise 23.

Let's assume that the distribution of letters in the plaintext closely matches that of English, as expressed by the vector A0 above. Look at a random letter in the top strip of ciphertext. It corresponds to a random letter of English shifted by some amount i (corresponding to an element of the key). The letter below it corresponds to a random letter of English shifted by some amount j.

For concreteness, let's suppose that i=0

and j=2. The probability that the letter in the 50th position, for example, is A is given by the first entry in A0, namely .082. The letter directly below, on the second strip, has been shifted from the original plaintext by j=2 positions. If this ciphertext letter is A, then the corresponding plaintext letter was y, which occurs in the plaintext with probability .020. Note that .020 is the first entry of the vector A2. The probability that the letter in the 50th position on the first strip and the letter directly below it are both the letter A is (.082)(.020). Similarly, the probability that both letters are B is (.015)(.001). Working all the way through Z, we see that the probability that the two letters are the same is

(.082)(.020)+(.015)(.001)+⋯+(.001)(.001)=A0·A2.

In general, when the encryption shifts are i and j, the probability that the two letters are the same is Ai·Aj. When $i \neq j$, this is approximately 0.038, but if i=j, then the dot product is 0.066.

We are in the situation where i=j exactly when the letters lying one above the other have been shifted by the same amount during the encryption process, namely when the top strip is displaced by an amount equal to the key length (or a multiple of the key length). Therefore we expect more coincidences in this case.

For a displacement of 5 in the preceding ciphertext, we had 326 comparisons and 24 coincidences. By the reasoning just

given, we should expect approximately $326 \times 0.066 = 21.5$ coincidences, which is close to the actual value.

2.3.3 Finding the Key: Second Method

Using the preceding ideas, we give another method for determining the key. It seems to work somewhat better than the first method on short samples, though it requires a little more calculation.

We'll continue to work with the preceding example. To find the first element of the key, count the occurrences of the letters in the 1st, 6th, 11th, ... positions, as before, and put them in a vector:

V=(0, 0, 7, 1, 1, 2, 9, 0, 1, 8, 8, 0, 0, 3, 0, 4, 5, 2, 0, 3, 6, 5, 1, 0, 1, 0)

(the first entry gives the number of occurrences of A, the second gives the number of occurrences of B, etc.). If we divide by 67, which is the total number of letters counted, we obtain a vector

W=(0, 0, .1045, .0149, .0149, .0299, …, .0149, 0).

Let's think about where this vector comes from. Since we know the key length is 5, the 1st, 6th, 11th, ... letters in the ciphertext were all shifted by the same amount (as we'll see shortly, they were all shifted by 2). Therefore, they represent a random sample of English letters, all shifted by the same amount. Their frequencies, which are given by the vector W, should approximate the vector Ai, where i is the shift caused by

the first element of the key.

The problem now is to determine i. Recall that Ai·Aj is largest when i=j, and that **W** approximates Ai. If we compute W·Aj for $0 \le j \le 25$, the maximum value should occur when j=i. Here are the dot products:

.0250, .0391, .0713, .0388, .0275, .0380, .0512, .0301, .0325, .0430, .0338, .02

The largest value is the third, namely .0713, which equals W·A2. Therefore, we guess that the first shift is 2, which corresponds to the key letter c.

Let's use the same method to find the third element of the key. We calculate a new vector **W**, using the frequencies for the 3rd, 8th, 13th, ... letters that we tabulated previously:

W=(0, .0152, 0, .0454, .0454, .0152, …, 0, .0152).

The dot products W·Ai for 0 ≤ i ≤ 25 are

.0372, .0267, .0395, .0624, .04741, .0279, .0319, .0504, .0378, .0351, .0367, .0

The largest of these values is the fourth, namely .0624, which equals W·A3. Therefore, the best guess is that the first shift is 3, which corresponds to the key letter d. The other three elements of the key can be found similarly, again yielding c, o, d, e, s as the key.

Notice that the largest dot product was significantly larger than the others in both cases, so we didn't have to make several guesses to find the correct one. In this way, the present method is superior to the first method presented; however, the first method is much easier to do by hand.

Why is the present method more accurate than the first one? To obtain the largest dot product, several of the larger values in **W** had to match with the larger values in an Ai. In the earlier method, we tried to match only the e, then looked at whether the choices for other letters were reasonable. The present method does this all in one step.

To summarize, here is the method for finding the key. Assume we already have determined that the key length is n.

For i=1 to n, do the following:

1. Compute the frequencies of the letters in positions i mod n, and form the vector **W**.

2. For j=0 to 25, compute W·Aj.

3. Let ki=j0 give the maximum value of W·Aj.

The key is probably {k1, …, kn}.

2.4 Substitution Ciphers

One of the more popular cryptosystems is the substitution cipher. It is commonly used in the puzzle section of the weekend newspapers, for example. The principle is simple: Each letter in the alphabet is replaced by another (or possibly the same) letter. More precisely, a permutation of the alphabet is chosen and applied to the plaintext. In the puzzle pages, the spaces between the words are usually preserved, which is a big advantage to the solver, since knowledge of word structure becomes very useful. However, to increase security it is better to omit the spaces.

The shift and affine ciphers are examples of substitution ciphers. The Vigenère cipher (see Section 2.3) is not, since it permutes blocks of letters rather than one letter at a time.

Everyone "knows" that substitution ciphers can be broken by frequency counts. However, the process is more complicated than one might expect.

Consider the following example. Thomas Jefferson has a potentially treasonous message that he wants to send to Ben Franklin. Clearly he does not want the British to read the text if they intercept it, so he encrypts it using a substitution cipher. Fortunately, Ben Franklin knows the permutation being used, so he can simply reverse the

permutation to obtain the original message (of course, Franklin was quite clever, so perhaps he could have decrypted it without previously knowing the key).

Now suppose we are working for the Government Code and Cypher School in England back in 1776 and are given the following intercepted message to decrypt.

LWNSOZBNWVWBAYBNVBSQWVWOHW

PBWZWMYPOBNPBBNWJPAWWRZSLWZ

BPIRYRPOIWRPQOWAIENBVBNPBPUSRE

JPRZBNWFYAVYIBSHNPFFIRWVVBNPBB

ESDWARUWRBVPAWIRVBIBYBWZPUSRE

BFSLWAVHASUBNWXSRVWRBSHBNWES

JHSAUSHESDWARUWRBQWXSUWVZWV

IVBNWAIENBSHBNWFWSFOWBSPOBWA

VBIBYBWRWLESDWARUWRBOPJIREIBV

RXIFOWVPRZSAEPRIKIREIBVFSLWAVIR

SVBOICWOJBSWHHWXBBNWIAVPHWBJ

A frequency count yields the following
(there are 520 letters in the text):

The approximate frequencies of letters in English were given in Section 2.3. We repeat some of the data here in Table 2.2. This allows us to guess with reasonable confidence that W represents e (though B is another possibility). But what about the other letters? We can guess that B, R, S, I, V, A, P, N, with maybe an exception or two, are

probably the same as t, a, o, i, n, s, h, r in some order. But a simple frequency count is not enough to decide which is which. What we need to do now is look at digrams, or pairs of letters. We organize our results in Table 2.3 (we only use the most frequent letters here, though it would be better to include all).

Table 2.2 Frequencies of Most Common Letters in English

e	t	a	o	i	n	s	h	r
.127	.091	.082	.075	.070	.067	.063	.061	.060

Table 2.2 Full Alternative Text

Table 2.3 Counting Digrams

	W	B	R	S	I	V	A	P	N
W	3	4	12	2	4	10	14	3	1
B	4	4	0	11	5	5	2	4	20
R	5	5	0	1	1	5	0	3	0
S	1	0	5	0	1	3	5	2	0
I	1	8	10	1	0	2	3	0	0
V	8	10	0	0	2	2	0	3	1
A	7	3	4	2	5	4	0	1	0
P	0	8	6	0	1	1	4	0	0
N	14	3	0	1	1	1	0	7	0

Table 2.3 Full Alternative Text

The entry 1 in the W row and N column means that the combination WN appears 1 time in the text. The entry 14 in the N row and W column means that NW appears 14 times.

We have already decided that W=e, but if we had extended the table to include low-frequency letters, we would see that W contacts many of these letters, too, which is another characteristic of e. This helps to confirm our guess.

The vowels a, i, o tend to avoid each

other. If we look at the R row, we see that R does not precede S, I, A, N very often. But a look at the R column shows that R follows S, I, A fairly often. So we suspect that R is not one of a, i, o. V and N are out because they would require a, i, or o to precede W=e quite often, which is unlikely. Continuing, we see that the most likely possibilities for a, i, o are S, I, P in some order.

The letter n has the property that around 80% of the letters that precede it are vowels. Since we already have identified W, S, I, P as vowels, we see that R and A are the most likely candidates. We'll have to wait to see which is correct.

The letter h often appears before e and rarely after it. This tells us that N=h.

The most common digram is th. Therefore, B=t.

Among the frequent letters, r and s remain, and they should equal V and one of A, R. Since r pairs more with vowels and s pairs more with consonants, we see that V must be s and r is represented by either A or R.

The combination rn should appear more than nr, and AR is more frequent than RA, so our guess is that A=r and R=n.

We can continue the analysis and determine that S=o (note that to is much more common than ot), I=i, and P=a are the most likely choices. We have therefore determined reasonable guesses for 382 of the 520 characters in the text:

At this point, knowledge of the language, middle-level frequencies (l, d, …), and educated guesses can be used to fill in the remaining letters. For example, in the first line a good guess is that Y=u since then the word truths appears. Of course, there is a lot of guesswork, and various hypotheses need to be tested until one works.

Since the preceding should give the spirit of the method, we skip the remaining details. The decrypted message, with spaces (but not punctuation) added, is as follows (the text is from the middle of the Declaration of Independence):

we hold these truths to be self evident that all men are created equal that they

are endowed by their creator with certain unalienable rights that among these are life liberty and the pursuit of happiness that to secure these rights governments are instituted among men deriving their just powers from the consent of the governed that whenever any form of government becomes destructive of these ends it is the right of the people to alter or to abolish it and to institute new government laying its foundation on such principles and organizing its powers in such form as to seem most likely to effect their safety and happiness

2.5 Sherlock Holmes

Cryptography has appeared in many places in literature, for example, in the works of Edgar Allen Poe (The Gold Bug), William Thackeray (The History of Henry Esmond), Jules Verne (Voyage to the Center of the Earth), and Agatha Christie (The Four Suspects).

Here we give a summary of an enjoyable tale by Arthur Conan Doyle, in which Sherlock Holmes displays his usual cleverness, this time by breaking a cipher system. We cannot do the story justice here, so we urge the reader to read The Adventure of the Dancing Men in its entirety. The following is a

cryptic, and cryptographic, summary of the plot.

Mr. Hilton Cubitt, who has recently married the former Elsie Patrick, mails Sherlock Holmes a letter. In it is a piece of paper with dancing stick figures that he found in his garden at Riding Thorpe Manor:

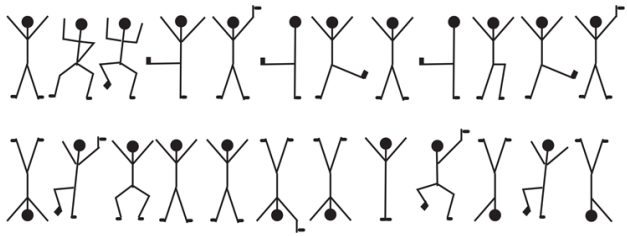
Two weeks later, Cubitt finds another series of figures written in chalk on his toolhouse door:

Two mornings later another sequence appears:

Three days later, another message appears:

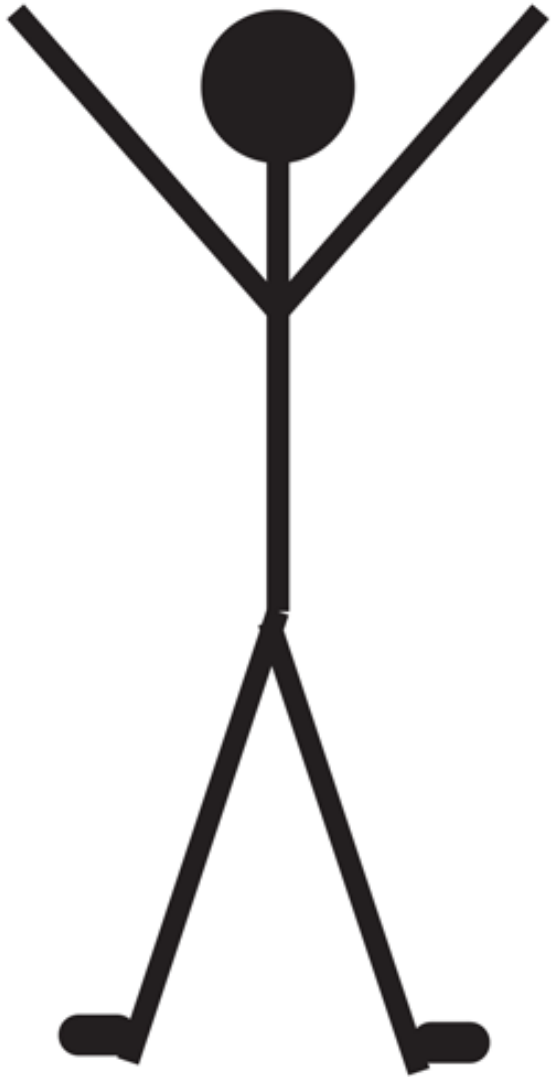
Cubitt gives copies of all of these to Holmes, who spends the next two days making many calculations. Suddenly, Holmes jumps from his chair, clearly having made a breakthrough. He quickly sends a long telegram to someone and then waits, telling Watson that they will probably be going to visit Cubitt the next day. But two days pass with no reply to the telegram, and then a letter arrives from Cubitt with yet another

message:



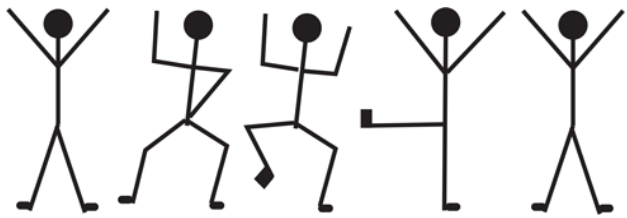
Holmes studies it and says they need to travel to Riding Thorpe Manor as soon as possible. A short time later, a reply to Holmes's telegram arrives, and Holmes indicates that the matter has become even more urgent. When Holmes and Watson arrive at Cubitt's house the next day, they find the police already there. Cubitt has been shot dead. His wife, Elsie, has also been shot and is in critical condition (although she survives). Holmes asks several questions and then has someone deliver a note to a Mr. Abe Slaney at nearby

Elrige's Farm. Holmes then explains to Watson and the police how he decrypted the messages. First, he guessed that the flags on some of the figures indicated the ends of words. He then noticed that the most common figure was



so it was likely E. This gave the fourth message as *–E–E–*. The possibilities *LEVER, NEVER, SEVER* came to mind, but since the message was probably a

one word reply to a previous message, Holmes guessed it was *NEVER*. Next, Holmes observed that



had the form *E–––E*, which could be *ELSIE*. The third message was therefore *–––E ELSIE*. Holmes tried several combinations, finally settling on *COME ELSIE* as the only viable possibility. The first message therefore was *–M – ERE ––E SL–NE–*. Holmes guessed that the first letter was A and the third letter as H, which gave the message as *AM HERE A–E SLANE–*. It was reasonable to complete this to *AM HERE ABE SLANEY*. The second

message then was *A– ELRI–ES*. Of course, Holmes correctly guessed that this must be stating where Slaney was staying. The only letters that seemed reasonable completed the phrase to *AT ELRIGES*. It was after decrypting these two messages that Holmes sent a telegram to a friend at the New York Police Bureau, who sent back the reply that Abe Slaney was “the most dangerous crook in Chicago.” When the final message arrived, Holmes decrypted it to *ELSIE –RE–ARE TO MEET THY GO–*. Since he recognized the missing letters as P, P, D, respectively, Holmes became very concerned and that’s why he decided to make the trip to Riding Thorpe Manor.

When Holmes finishes this explanation, the police urge that they go to Elrige's and arrest Slaney immediately. However, Holmes suggests that is unnecessary and that Slaney will arrive shortly. Sure enough, Slaney soon appears and is handcuffed by the police. While waiting to be taken away, he confesses to the shooting (it was somewhat in self-defense, he claims) and says that the writing was invented by Elsie Patrick's father for use by his gang, the Joint, in Chicago. Slaney was engaged to be married to Elsie, but she escaped from the world of gangsters and fled to London. Slaney finally traced her location and sent the secret messages. But why did Slaney walk into the trap that Holmes set? Holmes shows the

message he wrote:

From the letters already deduced, we see that this says *COME HERE AT ONCE*. Slaney was sure this message must have been from Elsie since he was certain no one outside of the Joint could write such messages. Therefore, he made the visit that led to his capture.

Comments

What Holmes did was solve a simple substitution cipher, though he did this with very little data. As with most such ciphers, both frequency analysis and a knowledge of the language are very useful. A little luck is nice, too, both in

the form of lucky guesses and in the distribution of letters. Note how overwhelmingly E was the most common letter. In fact, it appeared 11 times among the 38 characters in the first four messages. This gave Holmes a good start. If Elsie had been Carol and Abe Slaney had been John Smith, the decryption would probably have been more difficult.

Authentication is an important issue in cryptography. If Eve breaks Alice's cryptosystem, then Eve can often masquerade as Alice in communications with Bob. Safeguards against this are important. The judges gave Abe Slaney many years to think about this issue.

The alert reader might have noticed that

we cheated a little when decrypting the messages. The same symbol represents the V in *NEVER* and the *P*s in *PREPARE*. This is presumably due to a misprint and has occurred in every printed version of the work, starting with the story's first publication back in 1903. In the original text, the R in *NEVER* is written as the B in ABE, but this is corrected in later editions (however, in some later editions, the first C in the message Holmes wrote is given an extra arm and therefore looks like the M). If these mistakes had been in the text that Holmes was working with, he would have had a very difficult time decrypting and would have rightly concluded that the Joint needed to use error correction techniques in their

transmissions. In fact, some type of error correction should be used in conjunction with almost every cryptographic protocol.

2.6 The Playfair and ADFGX Ciphers

The Playfair and ADFGX ciphers were used in World War I by the British and the Germans, respectively. By modern standards, they are fairly weak systems, but they took real effort to break at the time.

The Playfair system was invented around 1854 by Sir Charles Wheatstone, who named it after his friend, the Baron Playfair of St. Andrews, who worked to convince the government to use it. In addition to being used in World War I, it was used by the British forces in the Boer War.

The key is a word, for example, *playfair*. The repeated letters are removed, to obtain *playfir*, and the remaining letters are used to start a 5×5 matrix. The remaining spaces in the matrix are filled in with the remaining letters in the alphabet, with i and j being treated as one letter:

playfirbcdeghkmnoqstuvwxz

Suppose the plaintext is *meet at the schoolhouse*. Remove spaces and divide the text into groups of two letters. If there is a doubled letter appearing as a group, insert an x and regroup. Add an extra x at the end to complete the last group, if necessary. Our plaintext becomes

me et at th es ch ox ol ho us ex.

Now use the matrix to encrypt each two-letter group by the following scheme:

- If the two letters are not in the same row or column, replace each letter by the letter that is in its row and is in the column of the other letter. For example, *et* becomes MN, since M is in the same row as e and the same column as t, and N is in the same row as t and the same column as e.
- If the two letters are in the same row, replace each letter with the letter immediately to its right, with the matrix wrapping around from the last column to the first. For example, *me* becomes EG.
- If the two letters are in the same column, replace each letter with the letter immediately below it, with the matrix wrapping around from the last row to the first. For example, ol becomes VR.

The ciphertext in our example is

EG MN FQ QM KN BK SV VR GQ XN KU.

To decrypt, reverse the procedure.

The system succumbs to a frequency

attack since the frequencies of the various digrams (two-letter combinations) in English have been tabulated. Of course, we only have to look for the most common digrams; they should correspond to the most common digrams in English: th, he, an, in, re, es, …. Moreover, a slight modification yields results more quickly. For example, both of the digrams re and er are very common. If the pairs IG and GI are common in the ciphertext, then a good guess is that e, i, r, g form the corners of a rectangle in the matrix. Another weakness is that each plaintext letter has only five possible corresponding ciphertext letters. Also, unless the keyword is long, the last few rows of the matrix are predictable.

Observations such as these allow the system to be broken with a ciphertext-only attack. For more on its cryptanalysis, see [Gaines].

The ADFGX cipher proceeds as follows. Put the letters of the alphabet into a 5×5 matrix. The letters i and j are treated as one, and the columns of the matrix are labeled with the letters A, D, F, G, X. For example, the matrix could be

	A	D	F	G	X
A	p	g	c	e	n
D	b	q	o	z	r
F	s	l	a	f	t
G	m	d	v	i	w
X	k	u	y	x	h

2.6-12 Full Alternative Text

Each plaintext letter is replaced by the label of its row and column. For example, s becomes FA, and z becomes DG. Suppose the plaintext is

Kaiser Wilhelm.

The result of this initial step is

XA FF GG FA AG DX GX GG FD XX AG FD GA.

So far, this is a disguised substitution cipher. The next step increases the complexity significantly. Choose a keyword, for example, *Rhein*. Label the columns of a matrix by the letters of the keyword and put the result of the initial step into another matrix:

R	H	E	I	N
X	A	F	F	G
G	F	A	A	G
D	X	G	X	G
G	F	D	X	X
A	G	F	D	G
A				

2.6-13 Full Alternative Text

Now reorder the columns so that the column labels are in alphabetic order:

E	H	I	N	R
F	A	F	G	X
A	F	A	G	G
G	X	X	G	D
D	F	X	X	G
F	G	D	G	A
				A

2.6-14 Full Alternative Text

Finally, the ciphertext is obtained by reading down the columns (omitting the labels) in order:

FAGDFAFXFGFAXXDGGGXGXGDGAA.

Decryption is easy, as long as you know the keyword. From the length of the keyword and the length of the ciphertext, the length of each column is determined. The letters are placed into columns, which are reordered to match the keyword. The original matrix is then used to recover the plaintext.

The initial matrix and the keyword were changed frequently, making cryptanalysis more difficult, since there was only a limited amount of ciphertext available for any combination. However, the system was successfully

attacked by the French cryptanalyst Georges Painvin and the Bureau du Chiffre, who were able to decrypt a substantial number of messages.

Here is one technique that was used. Suppose two different ciphertexts intercepted at approximately the same time agree for the first several characters. A reasonable guess is that the two plaintexts agree for several words. That means that the top few entries of the columns for one are the same as for the other. Search through the ciphertexts and find other places where they agree. These possibly represent the beginnings of the columns. If this is correct, we know the column lengths. Divide the ciphertexts into

columns using these lengths. For the first ciphertext, some columns will have one length and others will be one longer. The longer ones represent columns that should be near the beginning; the other columns should be near the end. Repeat for the second ciphertext. If a column is long for both ciphertexts, it is very near the beginning. If it is long for one ciphertext and not for the other, it goes in the middle. If it is short for both, it is near the end. At this point, try the various orderings of the columns, subject to these restrictions. Each ordering corresponds to a potential substitution cipher. Use frequency analysis to try to solve these. One should yield the plaintext, and the initial encryption matrix.

The letters ADFGX were chosen because their symbols in Morse code (·–, –··, ··–·, ––·, –··–) were not easily confused. This was to avoid transmission errors, and represents one of the early attempts to combine error correction with cryptography. Eventually, the ADFGX cipher was replaced by the ADFGVX cipher, which used a 6×6 initial matrix. This allowed all 26 letters plus 10 digits to be used.

For more on the cryptanalysis of the ADFGX cipher, see [Kahn].

2.7 Enigma

Mechanical encryption devices known as rotor machines were developed in the 1920s by several people. The best known was designed by Arthur Scherbius and became the famous Enigma machine used by the Germans in World War II.

It was believed to be very secure and several attempts at breaking the system ended in failure. However, a group of three Polish cryptologists, Marian Rejewski, Henryk Zygalski, and Jerzy Różycki, succeeded in breaking early versions of Enigma during the 1930s. Their techniques were passed to the

British in 1939, two months before Germany invaded Poland. The British extended the Polish techniques and successfully decrypted German messages throughout World War II.

The fact that Enigma had been broken remained a secret for almost 30 years after the end of the war, partly because the British had sold captured Enigma machines to former colonies and didn't want them to know that the system had been broken.

In the following, we give a brief description of Enigma and then describe an attack developed by Rejewski. For more details, see for example [Kozaczuk], which contains appendices by Rejeweski giving details of attacks

on Enigma.

We give a basic schematic diagram of the machine in Figure 2.1. For more details, we urge the reader to visit some of the many websites that can be found on the Internet that give pictures of actual Enigma machines and extensive diagrams of the internal workings of these machines. There are also several online Enigma simulators. Try one of them to get a better understanding of how Enigma works.

Figure 2.1 A Schematic Diagram of the Enigma Machine

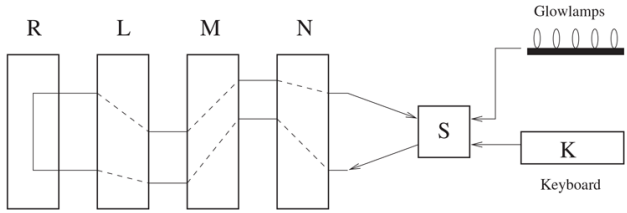


Figure 2.1 Full Alternative Text

L, M, N are the rotors. On one side of each rotor are 26 fixed electrical contacts, arranged in a circle. On the other side are 26 spring-loaded contacts, again arranged in a circle so as to touch the fixed contacts of the adjacent rotor. Inside each rotor, the fixed contacts are connected to the spring-loaded contacts in a somewhat random manner. These connections are different in each rotor. Each rotor has 26 possible initial settings.

R is the reversing drum. It has 26 spring-loaded contacts, connected in

pairs.

K is the keyboard and is the same as a typewriter keyboard.

S is the plugboard. It has approximately six pairs of plugs that can be used to interchange six pairs of letters.

When a key is pressed, the first rotor N turns 1/26 of a turn. Then, starting from the key, electricity passes through S, then through the rotors N, M, L. When it reaches the reversing drum R, it is sent back along a different path through L, M, N, then through S. At this point, the electricity lights a bulb corresponding to a letter on the keyboard, which is the letter of the ciphertext.

Since the rotor N rotates before each encryption, this is much more complicated than a substitution cipher. Moreover, the rotors L and M also rotate, but much less often, just like the wheels on a mechanical odometer.

Decryption uses exactly the same method. Suppose a sender and receiver have identical machines, both set to the same initial positions. The sender encrypts the message by typing it on the keyboard and recording the sequence of letters indicated by the lamps. This ciphertext is then sent to the receiver, who types the ciphertext into the machine. The sequence of letters appearing in the lamps is the original message. This can be seen as follows.

Lamp “a” and key “a” are attached to a wire coming out of the plugboard. Lamp “h” and key “h” are attached to another wire coming out of the plugboard. If the key “a” is pressed and the lamp “h” lights up, then the electrical path through the machine is also connecting lamp “a” to key “h”. Therefore, if the “h” key were pressed instead, then the “a” key would light.

Similar reasoning shows that no letter is ever encrypted as itself. This might appear to be a good idea, but actually it is a weakness since it allows a cryptanalyst to discard many possibilities at the start. See Chapter 14.

The security of the system rests on the keeping secret the initial settings of the

rotors, the setting of the plugs on the plugboard, and the internal wiring of the rotors and reversing drum. The settings of the rotors and the plugboard are changed periodically (for example, daily).

We'll assume the internal wiring of the rotors is known. This would be the case if a machine were captured, for example. However, there are ways to deduce this information, given enough ciphertext, and this is what was actually done in some cases.

How many combinations of settings are there? There are 26 initial settings for each of the three rotors. This gives 263=17576 possibilities. There are six possible orderings of the three rotors.

This yields 6×17576=105456 possible ways to initialize the rotors. In later versions of Enigma, there were five rotors available, and each day three were chosen. This made 60 possible orderings of the rotors and therefore 1054560 ways to initialize the rotors.

On the plugboard, there are 100391791500 ways of interchanging six pairs of letters.

In all, there seem to be too many possible initializations of the machine to have any hope of breaking the system. Techniques such as frequency analysis fail since the rotations of the rotors change the substitution for each character of the message.

So, how was Enigma attacked? We don't give the whole attack here, but rather show how the initial settings of the rotors were determined in the years around 1937. This attack depended on a weakness in the protocol being used at that time, but it gives the general flavor of how the attacks proceeded in other situations.

Each Enigma operator was given a codebook containing the daily settings to be used for the next month. However, if these settings had been used without modification, then each message sent during a given day would have had its first letter encrypted by the same substitution cipher. The rotor would then have turned and the second letter of

each text would have corresponded to another substitution cipher, and this substitution would have been the same for all messages for that day. A frequency analysis on the first letter of each intercepted message during a day would probably allow a decryption of the first letter of each text. A second frequency analysis would decrypt the second letters. Similarly, the remaining letters of the ciphertexts (except for the ends of the longest few ciphertexts) could be decrypted.

To avoid this problem, for each message the operator chose a message key consisting of a sequence of three letters, for example, r, f, u. He then used the daily setting from the codebook to

encrypt this message key. But since radio communications were prone to error, he typed in rfu twice, therefore encrypting rfurfu to obtain a string of six letters. The rotors were then set to positions r, f, and u and the encryption of the actual message began. So the first six letters of the transmitted message were the encrypted message key, and the remainder was the ciphertext. Since each message used a different key, frequency analysis didn't work.

The receiver simply used the daily settings from the codebook to decrypt the first six letters of the message. He then reset the rotors to the positions indicated by the decrypted message key and proceeded to decrypt the message.

The duplication of the key was a great aid to the cryptanalysts. Suppose that on some day you intercept several messages, and among them are three that have the following initial six letters:

dmqvbn

vonpuy

pucfmq

All of these were encrypted with the same daily settings from the codebook. The first encryption corresponds to a permutation of the 26 letters; let's call this permutation A. Before the second letter is encrypted, a rotor turns, so the second letter uses another permutation; call it B. Similarly, there are

permutations C, D, E, F for the remaining four letters. The strategy is to look at the products AD, BE, and CF.

We need a few conventions and facts about permutations. When we write AD for two permutations A and D, we mean that we apply the permutation A then D (some books use the reverse ordering). The permutation that maps a to b, b to c, and c to a will be denoted as the 3-cycle (abc). A similar notation will be used for cycles of other lengths. For example, (ab) is the permutation that switches a and b. A permutation can be written as a product of cycles. For example, the permutation

(dvpfkxgzyo)(eijmunqlht)(bc)(rw)(a)(s)

is the permutation that maps d to v, v to

p, t to e, r to w, etc., and fixes a and s. If the cycles are disjoint (meaning that no two cycles have letters in common), then this decomposition into cycles is unique.

Let's look back at the intercepted texts. We don't know the letters of any of the three message keys, but let's call the first message key xyz. Therefore, xyzxyz encrypts to dmqvbn. We know that permutation A sends x to d. Also, the fourth permutation D sends x to v. But we know more. Because of the internal wiring of the machine, A actually interchanges x and d and D interchanges x and v. Therefore, the product of the permutations, AD, sends d to v (namely, A sends d to x and then

D sends x to v). The unknown x has been eliminated. Similarly, the second intercepted text tells us that AD sends v to p, and the third tells us that AD sends p to f. We have therefore determined that

AD=(dvpf⋯)⋯.

In the same way, the second and fifth letters of the three messages tell us that

BE=(oumb⋯)⋯

and the third and sixth letters tell us that

CF=(cqny⋯)⋯.

With enough data, we can deduce the decompositions of AD, BE, and CF into products of cycles. For example, we might have

AD=(dvpfkxgzyo)(eijmunqlht)(bc)(rw)(a)(s)BE=(blfqveoum)(hjpswizrn)(axt)(cgy)(d)(k)CF=(abviktjgfcqny)(duzrehlxwpsmo).

This information depends only on the daily settings of the plugboard and the rotors, not on the message key. Therefore, it relates to every machine used on a given day.

Let's look at the effect of the plugboard. It introduces a permutation S at the beginning of the process and then adds the inverse permutation S−1 at the end. We need another fact about permutations: Suppose we take a permutation P and another permutation of the form SPS−1 for some permutation S (where S−1 denotes the inverse permutation of S; in our case, S=S−1) and decompose each into cycles. They will usually not have the

same cycles, but the lengths of the cycles in the decompositions will be the same. For example, AD has cycles of length 10, 10, 2, 2, 1, 1. If we decompose $SADS^{-1}$ into cycles for any permutation S, we will again get cycles of lengths 10, 10, 2, 2, 1, 1. Therefore, if the plugboard settings are changed, but the initial positions of the rotors remain the same, then the cycle lengths remain unchanged.

You might have noticed that in the decomposition of AD, BE, and CF into cycles, each cycle length appears an even number of times. This is a general phenomenon. For an explanation, see Appendix E of the aforementioned book by Kozaczuk.

Rejewski and his colleagues compiled a catalog of all 105456 initial settings of the rotors along with the set of cycle lengths for the corresponding three permutations AD, BE, CF. In this way, they could take the ciphertexts for a given day, deduce the cycle lengths, and find the small number of corresponding initial settings for the rotors. Each of these substitutions could be tried individually. The effect of the plugboard (when the correct setting was used) was then merely a substitution cipher, which was easily broken. This method worked until September 1938, when a modified method of transmitting message keys was adopted. Modifications of the above technique were again used to decrypt the messages. The process was also

mechanized, using machines called “bombes” to find daily keys, each in around two hours.

These techniques were extended by the British at Bletchley Park during World War II and included building more sophisticated “bombes.” These machines, designed by Alan Turing, are often considered to have been the first electronic computers.

2.8 Exercises

1. Caesar wants to arrange a secret meeting with Marc Antony, either at the Tiber (the river) or at the Coliseum (the arena). He sends the ciphertext EVIRE. However, Antony does not know the key, so he tries all possibilities. Where will he meet Caesar? (Hint: This is a trick question.)

2. Show that each of the ciphertexts ZOMCIH and ZKNGZR, which were obtained by shift ciphers from one-word plaintexts, has two different decryptions.

3. The ciphertext UCR was encrypted using the affine function 9x+2 mod 26. Find the plaintext.

4. The ciphertext JLH was obtained by affine encryption with the function 9x+1 mod 26. Find the plaintext.

5. Encrypt *howareyou* using the affine function 5x +7 (mod 26). What is the decryption function? Check that it works.

6. You encrypt messages using the affine function 9x

+2 mod 26. Decrypt the ciphertext GM.

7. A child has learned about affine ciphers. The parent says *NONONO*. The child responds with *hahaha*, and quickly claims that this is a decryption of the parent's message. The parent asks for the encryption function. What answer should the child give?

8. You try to encrypt messages using the affine cipher 4x+1 mod 26. Find two letters that encrypt to the same ciphertext letter.

9. The following ciphertext was encrypted by an affine cipher mod 26:

   CRWWZ.

   The plaintext starts *ha*. Decrypt the message.

10. Alice encrypts a message using the affine function $x \mapsto ax \pmod{26}$ for some a. The ciphertext is *FAP*. The third letter of the plaintext is T. Find the plaintext.

11. Suppose you encrypt using an affine cipher, then encrypt the encryption using another affine cipher (both are working mod 26). Is there any advantage to doing this, rather than using a single affine cipher? Why or why not?

12. Find all affine ciphers mod 26 for which the decryption function equals the encryption function. (There are 28 of them.)

13. Suppose we work mod 27 instead of mod 26 for affine ciphers. How many keys are possible? What if we work mod 29?

14. The ciphertext *XVASDW* was encrypted using an affine function ax+1 mod 26. Determine a and decrypt the message.

15. Suppose that you want to encrypt a message using an affine cipher. You let a=0, b=1, …, z=25, but you also include ?=26, ;=27, "=28, !=29. Therefore, you use $x \mapsto \alpha x+\beta \pmod{30}$ for your encryption function, for some integers α and β.

    1. Show that there are exactly eight possible choices for the integer α (that is, there are only eight choices of α (with $0<\alpha<30$) that allow you to decrypt).

    2. Suppose you try to use $\alpha=10$, $\beta=0$. Find two plaintext letters that encrypt to the same ciphertext letter.

16. You are trying to encrypt using the affine function 13x+22 mod 26.

1. Encrypt *HATE* and *LOVE*. Why is decryption impossible?

2. Find two different three-letter words that encrypt to *WWW*.

3. Challenge: Find a word (that is legal in various word games) that encrypts to *JJJ*. (There are four such words.)

17. You want to carry out an affine encryption using the function $\alpha x+\beta$, but you have $\gcd(\alpha, 26)=d>1$. Show that if $x1=x2+(26/d)$, then $\alpha x1+\beta \equiv \alpha x2+\beta \pmod{26}$. This shows that you will not be able to decrypt uniquely in this case.

18. You encrypt the message zzzzzzzzzz (there are 10 z's) using the following cryptosystems:

    1. affine cipher

    2. Vigenère cipher with key length 7

    Eve intercepts the ciphertexts. She knows the encryption methods (including key size) and knows what your plaintext is (she can hear you snoring). For each of the two cryptosystems, determine whether or not Eve can use this information to determine the key.

Explain your answer.

19. Suppose there is a language that has only the letters a and b. The frequency of the letter a is .1 and the frequency of b is .9. A message is encrypted using a Vigenère cipher (working mod 2 instead of mod 26). The ciphertext is BABABAAABA. The key length is 1, 2, or 3.

    1. Show that the key length is probably 2.

    2. Using the information on the frequencies of the letters, determine the key and decrypt the message.

20. Suppose you have a language with only the three letters a, b, c, and they occur with frequencies .9, .09, and .01, respectively. The ciphertext *BCCCBCBCBC* was encrypted by the Vigenère method (shifts are mod 3, not mod 26). Find the plaintext (Note: The plaintext is not a meaningful English message.)

21. Suppose you have a language with only the three letters a, b, c, and they occur with frequencies .7, .2, .1, respectively. The following ciphertext was encrypted by the Vigenère method (shifts are mod 3 instead of mod 26, of course):

ABCBABBBAC.

Suppose you are told that the key length is 1, 2, or 3. Show that the key length is probably 2, and determine the most probable key.

22. Victor designs a cryptosystem (called "Vector") as follows: He writes the letters in the plaintext as numbers mod 26 (with a=0, b=1, etc.) and groups them five at a time into five-dimensional vectors. His key is a five-dimensional vector. The encryption is adding the key vector mod 26 to each plaintext vector (so this is a shift cipher with vectors in place of individual letters).

    1. Describe a chosen plaintext attack on this system. Give the *explicit* plaintext used and how you get the key from the information you obtain.

    2. Victor's system is not new. It is the same as what well-known system?

23. If **v** and **w** are two vectors in n-dimensional space, $v \cdot w = |v||w| \cos\theta$, , where θ is the angle between the two vectors (measured in the two-dimensional plane spanned by the two vectors), and $|v|$ denotes the length of **v**. Use this fact to show that, in the notation of Section 2.3, the dot product $A0 \cdot Ai$ is largest when i=0.

24. Alice uses an improvement of the Vigenère cipher. She chooses five affine functions

    $$a_1x+b_1,\ a_2x+b_2,\ \ldots,\ a_5x+b_5 \pmod{26}$$

    and she uses these to encrypt in the style of Vigenère. Namely, she encrypts the first plaintext letter using a_1x+b_1, the second letter using a_2x+b_2, etc.

    1. What condition do $a_1, a_2, \ldots, a_5$ need to satisfy for Bob (who knows the key) to able to decrypt the message?

    2. Describe how to do a *chosen plaintext* attack to find the key. Give the plaintext *explicitly* and explain how it yields the key. (Note: the solution has nothing to do with frequencies of letters.)

25. Alice is sending a message to Bob using one of the following cryptosystems. In fact, Alice is bored and her plaintext consists of the letter a repeated a few hundred times. Eve knows what system is being used, but not the key, and intercepts the ciphertext. For systems (a), (b), and (c), state how Eve will recognize that the plaintext is one repeated letter and decide whether or not Eve can deduce the letter and the key.

1. Shift cipher

2. Affine cipher

3. Vigenère cipher

26. The operator of a Vigenère encryption machine is bored and encrypts a plaintext consisting of the same letter of the alphabet repeated several hundred times. The key is a seven-letter English word. Eve knows that the key is a word but does not yet know its length.

    1. What property of the ciphertext will make Eve suspect that the plaintext is one repeated letter and will allow her to guess that the key length is seven?

    2. Once Eve guesses that the plaintext is one repeated letter, how can she determine the key? (Hint: You need the fact that no English word of length seven is a shift of another English word.)

    3. Suppose Eve doesn't notice the property needed in part (a), and therefore uses the method of displacing then counting matches for finding the length of the key. What will the number of matches be for the various displacements? In other

words, why will the length of the key become very obvious by this method?

27. Use the Playfair cipher with the keyword *Cryptography* to encrypt

    Did he play fair at St Andrews golf course.

28. The ciphertext

    BPEGFCAIMAMGPOKBHU

    was encrypted using the Playfair cipher with keyword *Archimedes*. Find the plaintext.

29. Encrypt the plaintext *secret* using the ADFGX cipher with the 5×5 matrix in Section 2.6 and the keyword *spy*.

30. The ciphertext
    AAAAFXGGFAFFGGFGXAFGADGGAXXXFX
    was encrypted using the ADFGX cipher with the 5×5 matrix in Section 2.6 and the keyword *broken*. Find the plaintext.

31. Suppose Alice and Bob are using a cryptosystem with a 128-bit key, so there are 2128 possible keys. Eve is trying a brute-force attack on the system.

    1. Suppose it takes 1 day for Eve to try 264 possible keys. At this rate, how long will

it take for Eve to try all 2128 keys? (Hint: The answer is not 2 days.)

2. Suppose Alice waits 10 years and then buys a computer that is 100 times faster than the one she now owns (so it takes only 1/100 of a day, which is 864 seconds, to try 264 keys). Will she finish trying all 2128 keys before or after what she does in part (a)? (Note: This is a case where Aesop's Fable about the Tortoise and the Hare has a different ending.)

32. In the mid-1980s, a recruiting advertisement for NSA had 1 followed by one hundred 0s at the top. The text began "You're looking at a 'googol.' Ten raised to the 100th power. One followed by 100 zeroes. Counting 24 hours a day, you would need 120 years to reach a googol. Two lifetimes. It's a number that's impossible to grasp. A number beyond our imagination."

    How many numbers would you have to count each second in order to reach a googol in 120 years? (This problem is not related to the cryptosystems in this chapter. It is included to show how big 100-digit numbers are from a computational viewpoint. Regarding the ad, one guess is that the advertising firm assumed that the time it took to factor a 100-

digit number back then was the same as the time it took to count to a googol.)

2.9 Computer Problems

1. The following ciphertext was encrypted by a shift cipher:

   ```
   ycvejqwvhqtdtwvwu
   ```

   Decrypt. (The ciphertext is stored in the downloadable computer files (`bit.ly/2JbcS6p`) under the name *ycve*.)

2. The following ciphertext was the output of a shift cipher:

   ```
   lcllewljazlnnzmvyiylhrmhza
   ```

   By performing a frequency count, guess the key used in the cipher. Use the computer to test your hypothesis. What is the decrypted plaintext? (The ciphertext is stored in the downloadable computer files (`bit.ly/2JbcS6p`) under the name *lcll*.)

3. The following was encrypted by an affine cipher: `jidfbidzzteztxjsichfoihuszzsfsaichbipahsihzsichjujgfabbczggjsvzubehhgjsv`. Decrypt it. (This quote (NYTimes, 12/7/2014) is by Mark Wahlberg from when he was observing college classes in order to play a professor in “The Gambler." The ciphertext is stored in the downloadable computer files (`bit.ly/2JbcS6p`) under the name *jidf*.) (Hint: The command “frequency” could be useful. The plaintext has 9 e’s, 3 d’s, and 3 w’s.)

4. The following ciphertext was encrypted by an affine cipher:

   ```
   edsgickxhuklzveqzvkxwkzukcvuh
   ```

   The first two letters of the plaintext are *if*. Decrypt. (The ciphertext is stored in the downloadable computer files (`bit.ly/2JbcS6p`) under the name *edsg*.)

5. The following ciphertext was encrypted by an affine cipher using the function 3x+b for some b:

   ```

   ```

```
tcabtiqmfheqqmrmvmtmaq
```

Decrypt. (The ciphertext is stored in the downloadable computer files (`bit.ly/2JbcS6p`) under the name *tcab*.)

6. Experiment with the affine cipher $y \equiv mx+n \pmod{26}$ for values of $m>26$. In particular, determine whether or not these encryptions are the same as ones obtained with $m<26$.

7. In this problem you are to get your hands dirty doing some programming. Write some code that creates a new alphabet {A, C, G, T}. For example, this alphabet could correspond to the four nucleotides adenine, cytosine, guanine, and thymine, which are the basic building blocks of DNA and RNA codes. Associate the letters A, C, G, T with the numbers 0, 1, 2, 3, respectively.

    1. Using the shift cipher with a shift of 1, encrypt the following sequence of nucleotides, which is taken from the beginning of the thirteenth human chromosome:

       GAATTCGCGGCCGCAATTAACCCTCACTAAAGG

       CTAGAACT.

2. Write a program that performs affine ciphers on the nucleotide alphabet. What restrictions are there on the affine cipher?

8. The following was encrypted using by the Vigenère method using a key of length at most 6. Decrypt it and decide what is unusual about the plaintext. How did this affect the results?

```
hdsfgvmkoowafweetcmfthskucaqbilgjofmaql
mvswrvnqlszdmgaoqsakmlupsqforvtwvdfcjzv
vbksarlscdcaarmnvrysywxqgvellcyluwwveoa
ksepsoywxafowlbfcsocylngqsyzxgjbmlvgrgg
nrvqzcrggcrghgeupcyfgtydycjkhqluhgxgzov
sgazmyuhgsfhmftayjxmwznrsofrsoaopgauaaa
```

(The ciphertext is stored under the name *hdsf* in the downloadable computer files (`bit.ly/2JbcS6p`). The plaintext is from Gadsby by Ernest Vincent Wright.)

9. The following was encrypted by the Vigenère method. Find the plaintext.

```
ocwyikoooniwugpmxwktzdwgtssayjzwyemdlbn
oubfgqhgcscmgzlatoedcsdeidpbhtmuovpieki
xmpgkccaykwfzpyuavtelwhrhmwkbbvgtguvtef
```

```
tajbsauhzrzalkwuowhgedefnswmrciwcpaaavo
psjyeatcuceesohhdarkhwotikbroqrdfmzghgu
daylooqdmuhbdqgmyweuik
```

(The ciphertext is stored under the name *ocwy* in the downloadable computer files (`bit.ly/2JbcS6p`). The plaintext is from The Adventure of the Dancing Men by Sir Arthur Conan Doyle.)

10. The following was encrypted by the Vigenère method. Decrypt it. (The ciphertext is stored under the name *xkju* in the downloadable computer files (`bit.ly/2JbcS6p`).)

```
xkjurowmllpxwznpimbvbqjcnowxpcchhvvfvsl
axelxzxmyjaqfstsrulhhucdskbxknjqidallpq
hwhwewthbtxrljnrsncihuvffuxvoukjljswmaq
jultucpzmpliwmlubzxvoodybafdskxgqfadshx
saafvulluwtaqfrupwjrszxgpfutjqiynrxnynt
shyonddzzntzmplilrwnmwmlvuryonthuhabwnv
```

Chapter 3 Basic Number Theory

In modern cryptographic systems, the messages are represented by numerical values prior to being encrypted and transmitted. The encryption processes are mathematical operations that turn the input numerical values into output numerical values. Building, analyzing, and attacking these cryptosystems requires mathematical tools. The most important of these is number theory, especially the theory of congruences. This chapter presents the basic tools needed for the rest of the book. More advanced topics such as factoring, discrete logarithms, and elliptic curves,

will be treated in later chapters (Chapters 9, 10, and 21, respectively).

3.1 Basic Notions

3.1.1 Divisibility

Number theory is concerned with the properties of the integers. One of the most important is divisibility.

Definition

Let a and b be integers with $a \neq 0$. We say that a **divides** b, if there is an integer k such that b=ak. This is denoted by a|b. Another way to express this is that b is a multiple of a.

Example

$3|15,\ -15|60,\ 7\nmid 18$ (does not divide).

The following properties of divisibility are useful.

Proposition

Let a, b, c represent integers.

1. For every $a \neq 0$, $a|0$ and $a|a$. Also, $1|b$ for every b.
2. If $a|b$ and $b|c$, then $a|c$.
3. If $a|b$ and $a|c$, then $a|(sb+tc)$ for all integers s and t.

Proof. Since $0=a\cdot 0$, we may take $k=0$ in the definition to obtain $a|0$. Since $a=a\cdot 1$, we take $k=1$ to prove $a|a$. Since $b=1\cdot b$, we have $1|b$. This proves (1). In (2), there exist k and ℓ such that $b=ak$ and $c=b\ell$. Therefore, $c=(k\ell)a$, so $a|c$. For (3), write $b=ak1$ and $c=ak2$. Then sb

+tc=a(sk1+tk2), so a|sb+tc.

For example, take a=2 in part (2). Then 2|b simply means that b is even. The statement in the proposition says that c, which is a multiple of the even number b, must also be even (that is, a multiple of a=2).

3.1.2 Prime Numbers

A number p>1 whose positive divisors are only 1 and itself is called a **prime number**. The first few primes are 2, 3, 5, 7, 11, 13, 17, ⋯. An integer n>1 that is not prime is called **composite**, which means that n must be expressible as a product ab of integers with 1<a, b<n. A fact, known already to Euclid, is that there are infinitely many prime

numbers. A more precise statement is the following, proved in 1896.

Prime Number Theorem

Let $\pi(x)$ be the number of primes less than x. Then

$$\pi(x) \approx x \ln x,$$

in the sense that the ratio $\pi(x)/(x/\ln x) \to 1$ as $x \to \infty$.

We won't prove this here; its proof would lead us too far away from our cryptographic goals. In various applications, we'll need large primes, say of around 300 digits. We can estimate the number of 300-digit primes as follows:

$$\pi(10300) - \pi(10299) \approx 10300 \ln 10300 - 10299 \ln 10299 \approx 1.4 \times 10297.$$

So there are certainly enough such primes. Later, we'll discuss how to find them.

Prime numbers are the building blocks of the integers. Every positive integer has a unique representation as a product of prime numbers raised to different powers. For example, 504 and 1125 have the following factorizations:

$504 = 2^3 3^2 7$, $1125 = 3^2 5^3$.

Moreover, these factorizations are unique, except for reordering the factors. For example, if we factor 504 into primes, then we will always obtain three factors of 2, two factors of 3, and one factor of 7. Anyone who obtains the prime 41 as a factor has made a mistake.

Theorem

Every positive integer is a product of primes. This factorization into primes is unique, up to reordering the factors.

Proof. There is a small technicality that must be dealt with before we begin. When dealing with products, it is convenient to make the convention that an empty product equals 1. This is similar to the convention that x0=1. Therefore, the positive integer 1 is a product of primes, namely the empty product. Also, each prime is regarded as a one-factor product of primes.

Suppose there exist positive integers that are not products of primes. Let n be the smallest such integer. Then n cannot

be 1 (= the empty product), or a prime (= a one-factor product), so n must be composite. Therefore, n=ab with 1<a, b<n. Since n is the smallest positive integer that is not a product of primes, both a and b are products of primes. But a product of primes times a product of primes is a product of primes, so n=ab is a product of primes. This contradiction shows that the set of integers that are not products of primes must be the empty set. Therefore, every positive integer is a product of primes.

The uniqueness of the factorization is more difficult to prove. We need the following very important property of primes.

Lemma

If p is a prime and p divides a product of integers ab, then either p|a or p|b. More generally, if a prime p divides a product $ab \cdots z$, then p must divide one of the factors a, b, …, z.

For example, when p=2, this says that if a product of two integers is even then one of the two integers must be even. The proof of the lemma will be given at the end of the next section, after we discuss the Extended Euclidean algorithm.

Continuing with the proof of the theorem, suppose that an integer n can be written as a product of primes in two different ways:

n=p1a1p2a2⋯psas=q1b1q2b2⋯qtbt,

where p1, … , ps and q1, … , qt are primes, and the exponents ai and bj are nonzero. If a prime occurs in both factorizations, divide both sides by it to obtain a shorter relation. Continuing in this way, we may assume that none of the primes p1, … , ps occur among the qj's. Take a prime that occurs on the left side, say p1. Since p1 divides n, which equals q1q1⋯q1q2q2cdotsqtqt, the lemma says that p1 must divide one of the factors qj. Since qj is prime, p1=qj. This contradicts the assumption that p1 does not occur among the qj's. Therefore, an integer cannot have two distinct factorizations, as claimed.

3.1.3 Greatest Common

Divisor

The **greatest common divisor** of a and b is the largest positive integer dividing both a and b and is denoted by either gcd(a, b) or by (a, b). In this book, we use the first notation. To avoid technicalities, we always assume implicitly that at least one of a and b is nonzero.

Example

gcd(6, 4)=2, gcd(5, 7)=1, gcd(24, 60)=12.

We say that a and b are **relatively prime** if gcd(a, b)=1. There are two standard ways for finding the gcd:

1. If you can factor a and b into primes, do so. For each prime number, look at the powers that it appears in the factorizations of a and b. Take the

smaller of the two. Put these prime powers together to get the gcd. This is easiest to understand by examples:

576=2632, 135=335, gcd(576, 135)=32=9
gcd(253472, 22537)=22305071=227=28.

Note that if a prime does not appear in a factorization, then it cannot appear in the gcd.

2. Suppose a and b are large numbers, so it might not be easy to factor them. The gcd can be calculated by a procedure known as the **Euclidean algorithm**. It goes back to what everyone learned in grade school: division with remainder. Before giving a formal description of the algorithm, let's see some examples.

Example

Compute gcd(482, 1180).

SOLUTION

Divide 482 into 1180. The quotient is 2 and the remainder is 216. Now divide

the remainder 216 into 482. The quotient is 2 and the remainder is 50. Divide the remainder 50 into the previous remainder 216. The quotient is 4 and the remainder is 16. Continue this process of dividing the most recent remainder into the previous one. The last nonzero remainder is the gcd, which is 2 in this case:

$$1180=2\cdot 482+216482=2\cdot 216+50216=4\cdot 50+1650=3\cdot 16+216=8\cdot 2+0.$$

Notice how the numbers are shifted:

$$\text{remainder}\rightarrow\text{todivisor}\rightarrow\text{todividend}\rightarrow\text{toignore}.$$

Here is another example:

$$12345=1\cdot 11111+123411111=9\cdot 1234+51234=246\cdot 5+45=1\cdot 4+14=4\cdot 1+0.$$

Therefore, gcd(12345, 11111)=1.

Using these examples as guidelines, we

can now give a more formal description of the **Euclidean algorithm**. Suppose that a is greater than b. If not, switch a and b. The first step is to divide a by b, hence represent a in the form

$$a=q1b+r1.$$

If $r1=0$, then b divides a and the greatest common divisor is b. If $r1 \neq 0$, then continue by representing b in the form

$$b=q2r1+r2.$$

Continue in this way until the remainder is zero, giving the following sequence of steps:

$$a=q1b+r1b=q2r1+r2r1=q3r2+r3⋮rk−2=qkrk−1+rkrk−1=qk+1rk.$$

The conclusion is that

gcd(a, b)=rk.

There are two important aspects to this algorithm:

1. It does not require factorization of the numbers.
2. It is fast.

For a proof that it actually computes the gcd, see Exercise 59.

3.2 The Extended Euclidean Algorithm

The Euclidean Algorithm computes greatest common divisors quickly, but also, with only slightly more work, yields a very useful fact: gcd(a, b) can be expressed as a linear combination of a and b. That is, there exist integers x and y such that gcd(a, b)=ax+by. For example,

$$1=\gcd(45, 13)=45\cdot(-2)+13\cdot 77=\gcd(259, 119)=259\cdot 6-119\cdot 13.$$

The **Extended Euclidean Algorithm** will tell us how to find x and y. Rather than give a set of equations, we'll show how it works with the two examples we calculated in Subsection 3.1.3.

When we computed gcd(12345, 11111), we did the following calculation:

$$12345=1\cdot 11111+123411111=9\cdot 1234+51234=246\cdot 5+45=1\cdot 4+1.$$

For the Extended Euclidean Algorithm, we'll form a table with three columns and explain how they arise as we compute them.

We begin by forming two rows and three columns. The first entries in the rows are the original numbers we started with, namely 12345 and 11111. We will do some calculations so that we always have

$$\text{entry in first column} = 12345x+11111y,$$

where x and y are integers. The first two lines are trivial:

$12345 = 1 \cdot 12345 + 0 \cdot 11111$ and $11111 = 0 \cdot 12345 + 1 \cdot 11111$:

The first line in our gcd(12345, 11111) calculation tells us that $12345 = 1 \cdot 11111 + 1234$. We rewrite this as $1234 = 12345 - 1 \cdot 11111$. Using this, we compute

(1st row) −1·(2nd row),

yielding the following:

In effect, we have done the following

subtraction:

12345=12345(1)+11111(0)1111=12345(0)+11111(1)_1234=12345(0)+11111(

Therefore, the last line tells us that $1234=12345\cdot 1+11111\cdot(-1)$.

We now move to the second row of our gcd calculation. This says that $11111=9\cdot 1234+5$, which we rewrite as $5=11111-9\cdot 1234$. This tells us to compute (2nd row) $-9\cdot$(3rd row). We write this as

The last line tells us that $5=12345\cdot(-9)+11111\cdot10$.

The third row of our gcd calculation tells us that $4=1234-246\cdot5$. This becomes

Finally, we obtain

This tells us that $1 = 12345 \cdot (-2224) + 11111 \cdot 2471$.

Notice that as we proceeded, we were doing the Euclidean Algorithm in the first column. The first entry of each row is a remainder from the gcd calculation, and the entries in the second and third columns allow us to express the number in the first column as a linear combination of 12345 and 11111. The quotients in the Euclidean Algorithm tell us what to multiply a row by before

subtracting it from the previous row.

Let’s do another example using 482 and 1180 and our previous calculation that gcd(1180, 482)=2:

The end result is $2=1180\cdot(-29)+482\cdot71$.

To summarize, we state the following.

Theorem

Let a and b be integers with at least one of a, b nonzero. There exist integers x

and y, which can be found by the Extended Euclidean Algorithm, such that

$$\gcd(a, b)=ax+by.$$

As a corollary, we deduce the lemma we needed during the proof of the uniqueness of factorization into primes.

Corollary

If p is a prime and p divides a product of integers ab, then either $p|a$ or $p|b$. More generally, if a prime p divides a product $ab\cdots z$, then p must divide one of the factors a, b, … , z.

Proof. First, let's work with the case $p|ab$. If p divides a, we are done. Now assume $p\nmid a$. We claim $p|b$. Since p is

prime, gcd(a, p)=1 or p. Since $p \nmid a$, the gcd cannot be p. Therefore, gcd(a, p)=1, so there exist integers x, y with ax+py=1. Multiply by b to obtain abx+pby=b. Since p|ab and p|p, we have p|abx+pby, so p|b, as claimed.

If $p|ab \cdots z$, then p|a or $p|b \cdots z$. If p|a, we're done. Otherwise, $p|b \cdots z$. We now have a shorter product. Either p|b, in which case we're done, or p divides the product of the remaining factors. Continuing in this way, we eventually find that p divides one of the factors of the product.

The property of primes stated in the corollary holds only for primes. For example, if we know a product ab is divisible by 6, we cannot conclude that a

or b is a multiple of 6. The problem is that $6=2\cdot3$, and the 2 could be in a while the 3 could be in b, as seen in the example $60=4\cdot15$. More generally, if $n=ab$ is any composite, then $n|ab$ but $n\nmid a$ and $n\nmid b$. Therefore, the primes, and 1, are the only integers with the property of the corollary.

3.3 Congruences

One of the most basic and useful notions in number theory is modular arithmetic, or congruences.

Definition

Let a, b, n be integers with $n \neq 0$. We say that

$$a \equiv b \pmod{n}$$

(read: a is **congruent** to b mod n) if a−b is a multiple (positive or negative or zero) of n.

Another formulation is that $a \equiv b \pmod{n}$ if a and b differ by a multiple of n.

This can be rewritten as a=b+nk for some integer k (positive or negative).

Example

$32 \equiv 7 \pmod 5$, $-12 \equiv 37 \pmod 7$, $17 \equiv 17 \pmod{13}$.

Note: Many computer programs regard 17 (mod 10) as equal to the number 7, namely, the remainder obtained when 17 is divided by 10 (often written as 17%10=7). The notion of congruence we use is closely related. We have that two numbers are congruent mod n if they yield the same remainders when divided by n. For example, $17 \equiv 37 \pmod{10}$ because 17%10 and 37%10 are equal.

Congruence behaves very much like equality. In fact, the notation for

congruence was intentionally chosen to resemble the notation for equality.

Proposition

Let a, b, c, n be integers with $n \neq 0$.

1. $a \equiv 0 \pmod{n}$ if and only if $n|a$.
2. $a \equiv a \pmod{n}$.
3. $a \equiv b \pmod{n}$ if and only if $b \equiv a \pmod{n}$.
4. If $a \equiv b$ and $b \equiv c \pmod{n}$, then $a \equiv c \pmod{n}$.

Proof. In (1), $a \equiv 0 \pmod{n}$ means that $a=a-0$ is a multiple of n, which is the same as $n|a$. In (2), we have $a-a=0 \cdot n$, so $a \equiv a \pmod{n}$. In (3), if $a \equiv b \pmod{n}$, write $a-b=nk$. Then $b-a=n(-k)$, so $b \equiv a \pmod{n}$. Reversing the roles of a and b gives the reverse implication. For (4), write $a=b+nk$ and $b=c+n\ell$. Then a

$-c=n(k+\ell)$, so $a \equiv c \pmod{n}$.

Usually, we have $n>0$ and we work with the integers mod n, denoted Zn. These may be regarded as the set $\{0, 1, 2, \ldots, n-1\}$, with addition, subtraction, and multiplication mod n. If a is any integer, we may divide a by n and obtain a remainder in this set:

$a=nq+r$ with $0 \le r<n$.

(This is just division with remainder; q is the quotient and r is the remainder.) Then $a \equiv r \pmod{n}$, so every number a is congruent mod n to some integer r with $0 \le r<n$.

Proposition

Let a, b, c, d, n be integers with $n \neq 0$,

and suppose $a \equiv b \pmod{n}$ and $c \equiv d \pmod{n}$. Then

$a+c \equiv b+d$, $a-c \equiv b-d$, $ac \equiv bd \pmod{n}$.

Proof. Write $a=b+nk$ and $c=d+n\ell$, for integers k and ℓ. Then $a+c=b+d+n(k+\ell)$, so $a+c \equiv b+d \pmod{n}$. The proof that $a-c \equiv b-d$ is similar. For multiplication, we have $ac=bd+n(dk+b\ell+nk\ell)$, so $ac \equiv bd$.

The proposition says you can perform the usual arithmetic operations of addition, subtraction, and multiplication with congruences. You must be careful, however, when trying to perform division, as we'll see.

If we take two numbers and want to multiply them modulo n, we start by

multiplying them as integers. If the product is less than n, we stop. If the product is larger than n−1, we divide by n and take the remainder. Addition and subtraction are done similarly. For example, the integers modulo 6 have the following addition table:

+	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

3.3-1 Full Alternative Text

A table for multiplication mod 6 is

×	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	2	3	4	5
2	0	2	4	0	2	4
3	0	3	0	3	0	3
4	0	4	2	0	4	2
5	0	5	4	3	2	1

3.3-2 Full Alternative Text

Example

Here is an example of how we can do algebra mod n. Consider the following problem: Solve x+7 ≡ 3 (mod 17).

SOLUTION

x ≡ 3−7 ≡ −4 ≡ 13 (mod 17).

There is nothing wrong with negative answers, but usually we write the final answer as an integer from 0 to n−1

when we are working mod n.

3.3.1 Division

Division is much trickier mod n than it is with rational numbers. The general rule is that you can divide by a (mod n) when gcd(a, n)=1.

Proposition

Let a, b, c, n be integers with $n \neq 0$ and with gcd(a, n)=1. If $ab \equiv ac \pmod{n}$, then $b \equiv c \pmod{n}$. In other words, if a and n are relatively prime, we can divide both sides of the congruence by a.

Proof Since gcd(a, n)=1, there exist integers x, y such that ax+ny=1. Multiply by b−c to obtain

$(ab-ac)x+n(b-c)y=b-c.$

Since $ab-ac$ is a multiple of n, by assumption, and $n(b-c)y$ is also a multiple of n, we find that $b-c$ is a multiple of n. This means that $b \equiv c \pmod{n}$.

Example

Solve: $2x+7 \equiv 3 \pmod{17}$.

SOLUTION

$2x \equiv 3-7 \equiv -4$, so $x \equiv -2 \equiv 15 \pmod{17}$. The division by 2 is allowed since $\gcd(2, 17)=1$.

Example

Solve: $5x+6 \equiv 13 \pmod{11}$.

SOLUTION

$5x \equiv 7 \pmod{11}$. Now what do we do? We want to divide by 5, but what does 7/5 mean mod 11? Note that $7 \equiv 18 \equiv 29 \equiv 40 \equiv \cdots \pmod{11}$. So $5x \equiv 7$ is the same as $5x \equiv 40$. Now we can divide by 5 and obtain $x \equiv 8 \pmod{11}$ as the answer. Note that $7 \equiv 8 \cdot 5 \pmod{11}$, so 8 acts like 7/5.

The last example can be done another way. Since $5 \cdot 9 \equiv 1 \pmod{11}$, we see that 9 is the multiplicative inverse of 5 (mod 11). Therefore, dividing by 5 can be accomplished by multiplying by 9. If we want to solve $5x \equiv 7 \pmod{11}$, we multiply both sides by 9 and obtain

$x \equiv 45x \equiv 63 \equiv 8 \pmod{11}$.

Proposition

Suppose gcd(a, n)=1. Let s and t be integers such that as+nt=1 (they can be found using the extended Euclidean algorithm). Then $as \equiv 1 \pmod{n}$, so s is the multiplicative inverse for a (mod n).

Proof. Since as−1=−nt, we see that as −1 is a multiple of n.

Notation: We let a−1 denote this s, so a −1 satisfies $a^{-1}a \equiv 1 \pmod{n}$.

The extended Euclidean algorithm is fairly efficient for computing the multiplicative inverse of a by the method stated in the proposition.

Example

Solve $11111x \equiv 4 \pmod{12345}$.

SOLUTION

In Section 3.2, from the calculation of gcd(12345, 11111) we obtained

$$1 = 12345 \cdot (-2224) + 11111 \cdot 2471.$$

This says that

$$11111 \cdot 2471 \equiv 1 \pmod{12345}.$$

Multiplying both sides of the original congruence by 2471 yields

$$x \equiv 9884 \pmod{12345}.$$

In practice, this means that if we are working mod 12345 and we encounter the fraction 4/11111, we can replace it with 9884. This might seem a little strange, but think about what 4/11111

means. It's simply a symbol to represent a quantity that, when multiplied by 11111, yields 4. When we are working mod 12345, the number 9884 also has this property since $11111 \times 9884 \equiv 4 \pmod{12345}$.

Let's summarize some of the discussion:

Finding

$a^{-1} \pmod{n}$

1. Use the extended Euclidean algorithm to find integers s and t such that $as+nt=1$.

2. $a^{-1} \equiv s \pmod{n}$.

Solving

$ax \equiv c \pmod{n}$ **when** $\gcd(a, n)=1$

(Equivalently, you could be working

mod n *and encounter a fraction* c/a *with* gcd(a, n)=1.*)*

1. Use the extended Euclidean algorithm to find integers s and t such that as+nt=1.

2. The solution is $x \equiv cs \pmod{n}$ (equivalently, replace the fraction c/a with $cs \pmod{n}$).

What if

gcd(a, n)>1?

Occasionally we will need to solve congruences of the form $ax \equiv b \pmod{n}$ when gcd(a, n)=d>1. The procedure is as follows:

1. If d does not divide b, there is no solution.

2. Assume d|b. Consider the new congruence

   $(a/d)x \equiv b/d \pmod{n/d}$.

   Note that a/d, b/d, n/d are integers and gcd(a/d, n/d)=1. Solve this congruence by the above procedure to obtain a solution x0.

3. The solutions of the original congruence $ax \equiv b \pmod{n}$ are

$$x_0, x_0+(n/d), x_0+2(n/d), \ldots, x_0+(d-1)(n/d) \pmod{n}.$$

Example

Solve $12x \equiv 21 \pmod{39}$.

SOLUTION

$\gcd(12, 39)=3$, which divides 21. Divide by 3 to obtain the new congruence $4x \equiv 7 \pmod{13}$. A solution $x_0=5$ can be obtained by trying a few numbers, or by using the extended Euclidean algorithm. The solutions to the original congruence are $x \equiv 5, 18, 31 \pmod{39}$.

The preceding congruences contained x to the first power. However, nonlinear

congruences are also useful. In several places in this book, we will meet equations of the form

$x2 \equiv a(\text{mod } n)$.

First, consider $x2 \equiv 1 \text{ (mod } 7)$. The solutions are $x \equiv 1, 6 \text{ (mod } 7)$, as we can see by trying the values 0, 1, 2, … , 6 for x. In general, when p is an odd prime, $x2 \equiv 1 \text{ (mod } p)$ has exactly the two solutions $x \equiv \pm 1 \text{ (mod } p)$ (see Exercise 15).

Now consider $x2 \equiv 1 \text{ (mod } 15)$. If we try the numbers 0, 1, 2, … , 14 for x, we find that x=1, 4, 11, 14 are solutions. For example, $112 \equiv 121 \equiv 1 \text{ (mod } 15)$. Therefore, a quadratic congruence for a composite modulus can have more than two solutions, in contrast to the fact that

a quadratic equation with real numbers, for example, can have at most two solutions. In Section 3.4, we'll discuss this phenomenon. In Chapters 9 (factoring), 18 (flipping coins), and 19 (identification schemes), we'll meet applications of this fact.

3.3.2 Working with Fractions

In many situations, it will be convenient to work with fractions mod n. For example, $1/2 \pmod{12345}$ is easier to write than $6173 \pmod{12345}$ (note that $2 \times 6173 \equiv 1 \pmod{12345}$). The general rule is that a fraction b/a can be used mod n if gcd(a, n)=1. Of course, it should be remembered that $b/a \pmod{n}$ really means $a^{-1}b \pmod{n}$, where a^{-1}

denotes the integer mod n that satisfies $a^{-1}a \equiv 1 \pmod{n}$. But nothing will go wrong if it is treated as a fraction.

Another way to look at this is the following. The symbol "1/2" is simply a symbol with exactly one property: If you multiply 1/2 by 2, you get 1. In all calculations involving the symbol 1/2, this is the only property that is used. When we are working mod 12345, the number 6173 also has this property, since $6173 \times 2 \equiv 1 \pmod{12345}$. Therefore, $1/2 \pmod{12345}$ and $6173 \pmod{12345}$ may be used interchangeably.

Why can't we use fractions with arbitrary denominators? Of course, we cannot use $1/6 \pmod{6}$, since that

would mean dividing by $0 \pmod 6$. But even if we try to work with $1/2 \pmod 6$, we run into trouble. For example, $2 \equiv 8 \pmod 6$, but we cannot multiply both sides by 1/2, since $1 \not\equiv 4 \pmod 6$. The problem is that $\gcd(2, 6)=2 \neq 1$. Since 2 is a factor of 6, we can think of dividing by 2 as "partially dividing by 0." In any case, it is not allowed.

3.4 The Chinese Remainder Theorem

In many situations, it is useful to break a congruence mod n into a system of congruences mod factors of n. Consider the following example. Suppose we know that a number x satisfies $x \equiv 25 \pmod{42}$. This means that we can write $x=25+42k$ for some integer k. Rewriting 42 as $7 \cdot 6$, we obtain $x=25+7(6k)$, which implies that $x \equiv 25 \equiv 4 \pmod{7}$. Similarly, since $x=25+6(7k)$, we have $x \equiv 25 \equiv 1 \pmod{6}$. Therefore,

$$x \equiv 25 \pmod{42} \Rightarrow \begin{cases} x \equiv 4 \pmod{7} \\ x \equiv 1 \pmod{6}. \end{cases}$$

The Chinese remainder theorem shows that this process can be reversed;

namely, a system of congruences can be replaced by a single congruence under certain conditions.

Chinese Remainder Theorem

Suppose gcd(m, n)=1. Given integers a and b, there exists exactly one solution x (mod mn) to the simultaneous congruences

$$x \equiv a \pmod{m},\ x \equiv b \pmod{n}.$$

Proof. There exist integers s, t such that ms+nt=1. Then $ms \equiv 1 \pmod{n}$ and $nt \equiv 1 \pmod{m}$. Let x=bms+ant. Then $x \equiv ant \equiv a \pmod{m}$, and $x \equiv bms \equiv b \pmod{n}$, so a solution x exists. Suppose x1 is another solution. Then $x \equiv x1 \pmod{m}$ and $x \equiv x1 \pmod{n}$, so x−x1 is

a multiple of both m and n.

Lemma

Let m, n be integers with gcd(m, n)=1. If an integer c is a multiple of both m and n, then c is a multiple of mn.

Proof. Let $c=mk=n\ell$. Write $ms+nt=1$ with integers s, t. Multiply by c to obtain $c=cms+cnt=mn\ell s+mnkt=mn(\ell s+kt)$.

To finish the proof of the theorem, let $c=x-x1$ in the lemma to find that $x-x1$ is a multiple of mn. Therefore, $x \equiv x1 \pmod{mn}$. This means that any two solutions x to the system of congruences are congruent mod mn, as claimed.

Example

Solve $x \equiv 3 \pmod 7$, $x \equiv 5 \pmod{15}$.

SOLUTION

$x \equiv 80 \pmod{105}$ (note: $105=7\cdot 15$). Since $80 \equiv 3 \pmod 7$ and $80 \equiv 5 \pmod{15}$, 80 is a solution. The theorem guarantees that such a solution exists, and says that it is uniquely determined mod the product mn, which is 105 in the present example.

How does one find the solution? One way, which works with small numbers m and n, is to list the numbers congruent to $b \pmod n$ until you find one that is congruent to $a \pmod m$. For example, the numbers congruent to 5

(mod 15) are

5, 20, 35, 50, 65, 80, 95, ….

Mod 7, these are 5, 6, 0, 1, 2, 3, 4, …. Since we want 3 (mod 7), we choose 80.

For slightly larger numbers m and n, making a list would be inefficient. However, the proof of the theorem gives a fast method for finding x:

1. Use the Extended Euclidean algorithm to find s and t with ms+nt=1.

2. Let $x \equiv bms+ant \pmod{mn}$.

Example

Solve $x \equiv 7 \pmod{12345}$, $x \equiv 3 \pmod{11111}$.

SOLUTION

First, we know from our calculations in Section 3.2 that

$$12345\cdot(-2224)+11111\cdot 2471=1,$$

so $s=-2224$ and $t=2471$. Therefore,
$x \equiv 3\cdot 12345\cdot(-2224)+7\cdot 11111\cdot 2471 \equiv 10982112$
$(\mathrm{mod}\,(11111\cdot 12345))$.

How do you use the Chinese remainder theorem? The main idea is that if you start with a congruence mod a composite number n, you can break it into simultaneous congruences mod each prime power factor of n, then recombine the resulting information to obtain an answer mod n. The advantage is that often it is easier to analyze congruences mod primes or mod prime powers than to work mod composite numbers.

Suppose you want to solve $x2 \equiv 1 \pmod{35}$. Note that $35=5\cdot7$. We have

$$x2 \equiv 1 \pmod{35} \Leftrightarrow \{ x2 \equiv 1 \pmod{7} \; x2 \equiv 1 \pmod{5}.$$

Now, $x2 \equiv 1 \pmod{5}$ has two solutions: $x \equiv \pm 1 \pmod{5}$. Also, $x2 \equiv 1 \pmod{7}$ has two solutions: $x \equiv \pm 1 \pmod{7}$. We can put these together in four ways:

$$x \equiv 1 \pmod{5}, \; x \equiv 1 \pmod{7} \rightarrow x \equiv 1 \pmod{35}, \; x \equiv 1 \pmod{5}, \; x \equiv -1 \pmod{7} \rightarrow x \equiv 6 \pmod{35}, \; x \equiv -1 \pmod{5}, \; x \equiv 1 \pmod{7} \rightarrow x \equiv 29 \pmod{35}, \; x \equiv -1 \pmod{5}, \; x \equiv -1 \pmod{7} \rightarrow x \equiv 34 \pmod{35}.$$

So the solutions of $x2 \equiv 1 \pmod{35}$ are $x \equiv 1, 6, 29, 34 \pmod{35}$.

In general, if $n=p1p2 \cdots pr$ is the product of r distinct odd primes, then $x2 \equiv 1 \pmod{n}$ has $2r$ solutions. This is a consequence of the following.

Chinese Remainder Theorem (General Form)

Let $m_1, \ldots, m_k$ be integers with $\gcd(m_i, m_j)=1$ whenever $i \neq j$. Given integers $a_1, \ldots, a_k$, there exists exactly one solution $x \pmod{m_1 \cdots m_k}$ to the simultaneous congruences

$$x \equiv a_1 \pmod{m_1},\ x \equiv a_2 \pmod{m_2},\ \ldots,\ x \equiv a_k \pmod{m_k}.$$

For example, the theorem guarantees there is a solution to the simultaneous congruences

$$x \equiv 1 \pmod{11},\ x \equiv -1 \pmod{13},\ x \equiv 1 \pmod{17}.$$

In fact, $x \equiv 1871 \pmod{11 \cdot 13 \cdot 17}$ is the answer.

Exercise 57 gives a method for computing the number x in the theorem.

3.5 Modular Exponentiation

Throughout this book, we will be interested in numbers of the form

$x^a \pmod{n}$.

In this and the next couple of sections, we discuss some properties of numbers raised to a power modulo an integer.

Suppose we want to compute $21234 \pmod{789}$. If we first compute 21234, then reduce mod 789, we'll be working with very large numbers, even though the final answer has only 3 digits. We should therefore perform each multiplication and then calculate the remainder. Calculating the consecutive

powers of 2 would require that we perform the modular multiplication 1233 times. This is method is too slow to be practical, especially when the exponent becomes very large. A more efficient way is the following (all congruences are mod 789).

We start with $22 \equiv 4 \pmod{789}$ and repeatedly square both sides to obtain the following congruences:

$$24 \equiv 42 \equiv 1628 \equiv 162 \equiv 256216 \equiv 2562 \equiv 49232 \equiv 34264 \equiv 3672128 \equiv 5592256$$

Since 1234=1024+128+64+16+2 (this just means that 1234 equals 10011010010 in binary), we have

$$21234 \equiv 286 \cdot 559 \cdot 367 \cdot 49 \cdot 4 \equiv 481 \pmod{789}.$$

Note that we never needed to work with a number larger than 7882.

The same method works in general. If we want to compute $a^b \pmod n$, we can do it with at most $2\log_2(b)$ multiplications mod n, and we never have to work with numbers larger than n^2. This means that exponentiation can be accomplished quickly, and not much memory is needed.

This method is very useful if a, b, n are 100-digit numbers. If we simply computed a^b, then reduced mod n, the computer's memory would overflow: The number a^b has more than 10^{100} digits, which is more digits than there are particles in the universe. However, the computation of $a^b \pmod n$ can be accomplished in fewer than 700 steps by the present method, never using a

number of more than 200 digits.

Algorithmic versions of this procedure are given in Exercise 56. For more examples, see Examples 8 and 24–30 in the Computer Appendices.

3.6 Fermat's Theorem and Euler's Theorem

Two of the most basic results in number theory are Fermat's and Euler's theorems. Originally admired for their theoretical value, they have more recently proved to have important cryptographic applications and will be used repeatedly throughout this book.

Fermat's Theorem

If p is a prime and p does not divide a, then

$$a^{p-1} \equiv 1 \pmod{p}.$$

Proof. Let

$S=\{1, 2, 3, \ldots, p-1\}$.

Consider the map $\psi:S\to S$ defined by $\psi(x)=ax \pmod p$. For example, when $p=7$ and $a=2$, the map ψ takes a number x, multiplies it by 2, then reduces the result mod 7.

We need to check that if $x \in S$, then $\psi(x)$ is actually in S; that is, $\psi(x) \neq 0$. Suppose $\psi(x)=0$. Then $ax \equiv 0 \pmod p$. Since $\gcd(a, p)=1$, we can divide this congruence by a to obtain $x \equiv 0 \pmod p$, so $x \in S$. This contradiction means that $\psi(x)$ cannot be 0, hence $\psi(x) \in S$. Now suppose there are $x, y \in S$ with $\psi(x)=\psi(y)$. This means $ax \equiv ay \pmod p$. Since $\gcd(a, p)=1$, we can divide this congruence by a to obtain $x \equiv y \pmod p$. We conclude that if x, y are distinct

elements of S, then $\psi(x)$ and $\psi(y)$ are distinct. Therefore,

$$\psi(1), \psi(2), \psi(3), \cdots, \psi(p-1)$$

are distinct elements of S. Since S has only $p-1$ elements, these must be the elements of S written in a some order. It follows that

$$1 \cdot 2 \cdot 3 \cdots (p-1) \equiv \psi(1) \cdot \psi(2) \cdot \psi(3) \cdots \psi(p-1) \equiv (a \cdot 1)(a \cdot 2)(a \cdot 3) \cdots (a \cdot (p-1)) \equiv a^{p-1}(1 \cdot 2 \cdot 3 \cdots (p-1)) \pmod{p}.$$

Since $\gcd(j, p)=1$ for $j \in S$, we can divide this congruence by $1, 2, 3, \ldots, p-1$. What remains is $1 \equiv a^{p-1} \pmod{p}$.

Example

$2^{10}=1024 \equiv 1 \pmod{11}$. From this we can evaluate $2^{53} \pmod{11}$: Write $2^{53}=(2^{10})^5 2^3 \equiv 1^5 2^3 \equiv 8 \pmod{11}$. Note that when working mod 11, we are

essentially working with the exponents mod 10, not mod 11. In other words, from $53 \equiv 3 \pmod{10}$, we deduce $253 \equiv 23 \pmod{11}$.

The example leads us to a very important fact:

Basic Principle

Let p be prime and let a, x, y be integers with gcd(a, p)=1. If $x \equiv y \pmod{p-1}$, then $ax \equiv ay \pmod{p}$. In other words, if you want to work mod p, you should work mod p−1 in the exponent.

Proof. Write x=y+(p−1)k. Then

$$ax=ay+(p-1)k=ay(ap-1)k \equiv ay1k \equiv ay(\text{mod } p).$$

This completes the proof.

In the rest of this book, almost every time you see a congruence mod p−1, it will involve numbers that appear in exponents. The Basic Principle that was just stated shows that this translates into an overall congruence mod p. Do not make the (unfortunately, very common) mistake of working mod p in the exponent with the hope that it will yield an overall congruence mod p. It doesn't.

We can often use Fermat's theorem to show that a number is composite, without factoring. For example, let's show that 49 is composite. We use the technique of Section 3.5 to calculate

$$2^2 \equiv 4 \pmod{49}$$
$$2^4 \equiv 16^2 \equiv 16^2 \equiv 1121^6 \equiv 112 \equiv 2323^2 \equiv 232 \equiv 3924^8 \equiv 232^{216} \equiv 39 \cdot 23 \equiv$$

Since

$248 \equiv 1 (\mathrm{mod}\, 49)$,

we conclude that 49 cannot be prime (otherwise, Fermat's theorem would require that $248 \equiv 1 (\mathrm{mod}\, 49)$). Note that we showed that a factorization must exist, even though we didn't find the factors.

Usually, if $2n-1 \equiv 1 (\mathrm{mod}\, n)$, the number n is prime. However, there are exceptions: $561=3 \cdot 11 \cdot 17$ is composite but $2560 \equiv 1 (\mathrm{mod}\, 561)$. We can see this as follows: Since $560 \equiv 0 (\mathrm{mod}\, 2)$, we have $2560 \equiv 20 \equiv 1 (\mathrm{mod}\, 3)$. Similarly, since $560 \equiv 0 (\mathrm{mod}\, 10)$ and $560 \equiv 0 (\mathrm{mod}\, 16)$, we can conclude that $2560 \equiv 1 (\mathrm{mod}\, 11)$ and $2560 \equiv 1 (\mathrm{mod}\, 17)$. Putting things together via the Chinese remainder theorem, we find that

$2560 \equiv 1 \pmod{561}$.

Another such exception is $1729 = 7 \cdot 13 \cdot 19$. However, these exceptions are fairly rare in practice. Therefore, if $2^{n-1} \equiv 1 \pmod{n}$, it is quite likely that n is prime. Of course, if $2^{n-1} \not\equiv 1 \pmod{n}$, then n cannot be prime.

Since $2^{n-1} \pmod{n}$ can be evaluated very quickly (see Section 3.5), this gives a way to search for prime numbers. Namely, choose a starting point n_0 and successively test each odd number $n \geq n_0$ to see whether $2^{n-1} \equiv 1 \pmod{n}$. If n fails the test, discard it and proceed to the next n. When an n passes the test, use more sophisticated techniques (see Section 9.3) to test n for primality. The

advantage is that this procedure is much faster than trying to factor each n, especially since it eliminates many n quickly. Of course, there are ways to speed up the search, for example, by first eliminating any n that has small prime factors.

For example, suppose we want to find a random 300-digit prime. Choose a random 300-digit odd integer n0 as a starting point. Successively, for each odd integer $n \geq n0$, compute $2n-1 \pmod{n}$ by the modular exponentiation technique of Section 3.5. If $2n-1 \equiv 1 \pmod{n}$, Fermat's theorem guarantees that n is not prime. This will probably throw out all the composites encountered. When you find an n with

$2^{n-1} \equiv 1 \pmod{n}$, you probably have a prime number. But how many n do we have to examine before finding the prime? The Prime Number Theorem (see Subsection 3.1.2) says that the number of 300-digit primes is approximately 1.4×10^{297}, so approximately 1 out of every 690 numbers is prime. But we are looking only at odd numbers, so we expect to find a prime approximately every 345 steps. Since the modular exponentiations can be done quickly, the whole process takes much less than a second on a laptop computer.

We'll also need the analog of Fermat's theorem for a composite modulus n. Let $\phi(n)$ be the number of integers $1 \le a \le n$

such that gcd(a, n)=1. For example, if n=10, then there are four such integers, namely 1,3,7,9. Therefore, $\phi(10)=4$. Often ϕ is called **Euler's ϕ-function**.

If p is a prime and n=pr, then we must remove every pth number in order to get the list of a's with gcd(a, n)=1, which yields

$$\phi(pr)=(1-1p)pr.$$

In particular,

$$\phi(p)=p-1.$$

More generally, it can be deduced from the Chinese remainder theorem that for any integer n,

$$\phi(n)=n\prod p|n(1-1p),$$

where the product is over the distinct

primes p dividing n. When n=pq is the product of two distinct primes, this yields

$$\phi(pq)=(p-1)(q-1).$$

Examples

$\phi(10)=(2-1)(5-1)=4,$
$\phi(120)=120(1-12)(1-13)(1-15)=32$

Euler's Theorem

If gcd(a, n)=1, then

$$a^{\phi(n)} \equiv 1 \pmod{n}.$$

Proof. The proof of this theorem is almost the same as the one given for Fermat's theorem. Let S be the set of integers $1 \le x \le n$ with gcd(x, n)=1. Let $\psi:S\to S$ be defined by $\psi(x) \equiv ax \pmod{n}$. As in the proof of Fermat's theorem, the

numbers $\psi(x)$ for $x \in S$ are the numbers in S written in some order. Therefore,

$$\prod x \in Sx \equiv \prod x \in S\psi(x) \equiv a\phi(n) \prod x \in Sx.$$

Dividing out the factors $x \in S$, we are left with $1 \equiv a\phi(n) \pmod{n}$.

Note that when n=p is prime, Euler's theorem is the same as Fermat's theorem.

Example

What are the last three digits of 7803?

SOLUTION

Knowing the last three digits is the same as working mod 1000. Since $\phi(1000)=1000(1-12)(1-15)=400$, we

have $7803=(7400)273 \equiv 73 \equiv 343 \pmod{1000}$. Therefore, the last three digits are 343.

In this example, we were able to change the exponent 803 to 3 because $803 \equiv 3 \pmod{\phi(1000)}$.

Example

Compute $243210 \pmod{101}$.

SOLUTION

Note that 101 is prime. From Fermat's theorem, we know that $2100 \equiv 1 \pmod{101}$. Therefore,

$$243210 \equiv (2100)432210 \equiv 1432210 \equiv 1024 \equiv 14 \pmod{101}.$$

In this case, we were able to change the

exponent 43210 to 10 because $43210 \equiv 10 \pmod{100}$.

To summarize, we state the following, which is a generalization of what we know for primes:

Basic Principle

Let a, n, x, y be integers with $n \geq 1$ and $\gcd(a, n)=1$. If $x \equiv y \pmod{\phi(n)}$, then $ax \equiv ay \pmod{n}$. In other words, if you want to work mod n, you should work mod $\phi(n)$ in the exponent.

Proof. Write $x=y+\phi(n)k$. Then

$$ax=ay+\phi(n)k=ay(a\phi(n))k \equiv ay1k \equiv ay \pmod{n}.$$

This completes the proof.

This extremely important fact will be used repeatedly in the remainder of the book. Review the preceding examples until you are convinced that the exponents mod 400=$\phi(1000)$ and mod 100 are what count (i.e., don't be one of the many people who mistakenly try to work with the exponents mod 1000 and mod 101 in these examples).

3.6.1 Three-Pass Protocol

Alice wishes to transfer a secret key K (or any short message) to Bob via communication on a public channel. The Basic Principle can be used to solve this problem.

First, here is a nonmathematical way to do it. Alice puts K into a box and puts

her lock on the box. She sends the locked box to Bob, who puts his lock on the box and sends the box back to Alice. Alice then takes her lock off and sends the box to Bob. Bob takes his lock off, opens the box, and finds K.

Here is the mathematical realization of the method. First, Alice chooses a large prime number p that is large enough to represent the key K. For example, if Alice were trying to send a 56-bit key, she would need a prime number that is at least 56 bits long. However, for security purposes (to make what is known as the discrete log problem hard), she would want to choose a prime significantly longer than 56 bits. Alice publishes p so that Bob (or anyone else)

can download it. Bob downloads p. Alice and Bob now do the following:

1. Alice selects a random number a with gcd(a, p −1)=1 and Bob selects a random number b with gcd(b, p−1)=1. We will denote by a−1 and b−1 the inverses of a and b mod p−1.

2. Alice sends $K1 \equiv Ka \pmod{p}$ to Bob.

3. Bob sends $K2 \equiv K1b \pmod{p}$ to Alice.

4. Alice sends $K3 \equiv K2a-1 \pmod{p}$ to Bob.

5. Bob computes $K \equiv K3b-1 \pmod{p}$.

At the end of this protocol, both Alice and Bob have the key K.

The reason this works is that Bob has computed $Kaba-1b-1 \pmod{p}$. Since $aa-1 \equiv bb-1 \equiv 1 \pmod{p-1}$, the Basic Principle implies that $Kaba-1b-1 \equiv K1 \equiv K \pmod{p}$.

The procedure is usually attributed to Shamir and to Massey and Omura. One drawback is that it requires multiple communications between Alice and Bob. Also, it is vulnerable to the intruder-in-the-middle attack (see Chapter 15).

3.7 Primitive Roots

Consider the powers of 3 (mod 7):

$3^1 \equiv 3, 3^2 \equiv 2, 3^3 \equiv 6, 3^4 \equiv 4, 3^5 \equiv 5, 3^6 \equiv 1.$

Note that we obtain all the nonzero congruence classes mod 7 as powers of 3. This means that 3 is a primitive root mod 7 (the term *multiplicative generator* might be better but is not as common). Similarly, every nonzero congruence class mod 13 is a power of 2, so 2 is a primitive root mod 13. However, $3^3 \equiv 1 \pmod{13}$, so the powers of 3 mod 13 repeat much more frequently:

$3^1 \equiv 3, 3^2 \equiv 9, 3^3 \equiv 1, 3^4 \equiv 3, 3^5 \equiv 9, 3^6 \equiv 1, \ldots,$

so only 1, 3, 9 are powers of 3. Therefore, 3 is not a primitive root mod 13. The primitive roots mod 13 are 2, 6, 7, 11.

In general, when p is a prime, a **primitive root** mod p is a number whose powers yield every nonzero class mod p. It can be shown that there are $\phi(p-1)$ primitive roots mod p. In particular, there is always at least one. In practice, it is not difficult to find one, at least if the factorization of $p-1$ is known. See Exercise 54.

The following summarizes the main facts we need about primitive roots.

Proposition

Let α be a primitive root for the prime p.

1. Let n be an integer. Then $\alpha n \equiv 1 \pmod{p}$ if and only if $n \equiv 0 \pmod{p-1}$.

2. If j and k are integers, then $\alpha j \equiv \alpha k \pmod{p}$ if and only if $j \equiv k \pmod{p-1}$.

3. A number β is a primitive root mod p if and only if $p-1$ is the smallest positive integer k such that $\beta k \equiv 1 \pmod{p}$.

Proof. If $n \equiv 0 \pmod{p-1}$, then $n=(p-1)m$ for some m. Therefore,

$\alpha n \equiv (\alpha m)p-1 \equiv 1 \pmod{p}$

by Fermat's theorem. Conversely, suppose $\alpha n \equiv 1 \pmod{p}$. We want to show that $p-1$ divides n, so we divide $p-1$ into n and try to show that the remainder is 0. Write

$n=(p-1)q+r$, with $0 \leq r<p-1$

(this is just division with quotient q and remainder r). We have

$$1 \equiv \alpha n \equiv (\alpha q)p-1\alpha r \equiv 1\cdot\alpha r \equiv \alpha r (\mathrm{mod}\, p).$$

Suppose r>0. If we consider the powers $\alpha, \alpha 2, \ldots$ of $\alpha \,(\mathrm{mod}\, p)$, then we get back to 1 after r steps. Then

$$\alpha r+1 \equiv \alpha,\ \alpha r+2 \equiv \alpha 2, \ldots$$

so the powers of $\alpha \,(\mathrm{mod}\, p)$ yield only the r numbers $\alpha, \alpha 2, \ldots, 1$. Since $r<p-1$, not every number mod p can be a power of α. This contradicts the assumption that α is a primitive root.

The only possibility that remains is that r=0. This means that n=(p−1)q, so p−1 divides n. This proves part (1).

For part (2), assume that $j \geq k$ (if not,

switch j and k). Suppose that $\alpha j \equiv \alpha k \pmod{p}$. Dividing both sides by αk yields $\alpha j-k \equiv 1 \pmod{p}$. By part (1), $j-k \equiv 0 \pmod{p-1}$, so $j \equiv k \pmod{p-1}$. Conversely, if $j \equiv k \pmod{p-1}$, then $j-k \equiv 0 \pmod{p-1}$, so $\alpha j-k \equiv 1 \pmod{p}$, again by part (1). Multiplying by αk yields the result.

For part (3), if β is a primitive root, then part (1) says that any integer k with $\beta k \equiv 1$ modp must be a multiple of $p-1$, so $k=p-1$ is the smallest. Conversely, suppose $k=p-1$ is the smallest. Look at the numbers $1, \beta, \beta 2, \ldots, \beta p-2 \pmod{p}$. If two are congruent mod p, say $\beta i \equiv \beta j$ with $0 \leq i<j \leq p-2$, then $\beta j-i \equiv 1 \pmod{p}$ (note: $\beta p-1 \equiv 1 \pmod{p}$ implies that $\beta \equiv 0 \pmod{p}$, so we can divide by β).

Since $0<j-i<p-1$, this contradicts the assumption that $k=p-1$ is smallest. Therefore, the numbers $1, \beta, \beta 2, \ldots, \beta p-2$ must be distinct mod p. Since there are $p-1$ numbers on this list and there are $p-1$ numbers $1, 2, 3, \ldots, p-1$ mod p, the two lists must be the same, up to order. Therefore, each number on the list $1, 2, 3, \ldots, p-1$ is congruent to a power of β, so β is a primitive root mod p.

Warning: α is a primitive root mod p if and only if $p-1$ is the smallest positive n such that $\alpha n \equiv 1 \pmod{p}$. If you want to prove that α is a primitive root, it does not suffice to prove that $\alpha p-1 \equiv 1 \pmod{p}$. After all, Fermat's theorem says that every α satisfies this, as long as $\alpha \equiv 0$

(mod p). To prove that α is a primitive root, you must show that $p-1$ is the *smallest* positive exponent k such that $\alpha^k \equiv 1$.

3.8 Inverting Matrices Mod *n*

Finding the inverse of a matrix mod n can be accomplished by the usual methods for inverting a matrix, as long as we apply the rule given in Section 3.3 for dealing with fractions. The basic fact we need is that a square matrix is invertible mod n if and only if its determinant and n are relatively prime.

We treat only small matrices here, since that is all we need for the examples in this book. In this case, the easiest way is to find the inverse of the matrix is to use rational numbers, then change back to numbers mod n. It is a general fact that

the inverse of an integer matrix can always be written as another integer matrix divided by the determinant of the original matrix. Since we are assuming the determinant and n are relatively prime, we can invert the determinant as in Section 3.3.

For example, in the 2×2 case the usual formula is

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad-bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix},$$

so we need to find an inverse for $ad-bc \pmod{n}$.

Example

Suppose we want to invert $1234 \pmod{11}$. Since $ad-bc=-2$, we need the inverse of $-2 \bmod 11$. Since $5\times(-2) \equiv 1$

(mod 11), we can replace −1/2 by 5 and obtain

$$(1234)^{-1} \equiv -12(4^{-2}-31) \equiv 5(4^{-2}-31) \equiv (9175) \pmod{11}.$$

A quick calculation shows that

$$(1234)(9175) = (23115523) \equiv (1001) \pmod{11}.$$

Example

Suppose we want the inverse of

$$M = (111123149) \pmod{11}.$$

The determinant is 2 and the inverse of M in rational numbers is

$$126 - 51 - 68 - 22 - 31.$$

(For ways to calculate the inverse of a matrix, look at any book on linear algebra.) We can replace 1/2 with 6 mod 11 and obtain

$$M^{-1} \equiv (3368410146) \pmod{11}.$$

Why do we need the determinant and n to be relatively prime? Suppose $MN \equiv I \pmod{n}$, where I is the identity matrix. Then

$$\det(M)\det(N) \equiv \det(MN) \equiv \det(I) \equiv 1 \pmod{n}.$$

Therefore, det(M) has an inverse mod n, which means that det(M) and n must be relatively prime.

3.9 Square Roots Mod *n*

Suppose we are told that $x2 \equiv 71 \pmod{77}$ has a solution. How do we find one solution, and how do we find all solutions? More generally, consider the problem of finding all solutions of $x2 \equiv b \pmod{n}$, where $n=pq$ is the product of two primes. We show in the following that this can be done quite easily, once the factorization of n is known. Conversely, if we know all solutions, then it is easy to factor n.

Let's start with the case of square roots mod a prime p. The easiest case is when $p \equiv 3 \pmod{4}$, and this suffices for our purposes. The case when $p \equiv 1 \pmod{4}$

is more difficult. See [Cohen, pp. 31–34] or [KraftW, p. 317].

Proposition

Let $p \equiv 3 \pmod 4$ be prime and let y be an integer. Let $x \equiv y^{(p+1)/4} \pmod p$.

1. If y has a square root mod p, then the square roots of y mod p are ±x.

2. If y has no square root mod p, then −y has a square root mod p, and the square roots of −y are ±x.

Proof. If $y \equiv 0 \pmod p$, all the statements are trivial, so assume $y \equiv 0 \pmod p$. Fermat's theorem says that $y^{p-1} \equiv 1 \pmod p$. Therefore,

$$x^4 \equiv y^{p+1} \equiv y^{2y^{p-1}} \equiv y^2 \pmod p.$$

This implies that $(x^2+y)(x^2-y) \equiv 0$

(mod p), so $x^2 \equiv \pm y \pmod{p}$. (See Exercise 13(a).) Therefore, at least one of y and $-y$ is a square mod p. Suppose both y and $-y$ are squares mod p, say $y \equiv a^2$ and $-y \equiv b^2$. Then $-1 \equiv (a/b)^2$ (work with fractions mod p as in Section 3.3), which means -1 is a square mod p. This is impossible when $p \equiv 3 \pmod{4}$ (see Exercise 26). Therefore, exactly one of y and $-y$ has a square root mod p. If y has a square root mod p then $y \equiv x^2$, and the two square roots of y are $\pm x$. If $-y$ has a square root, then $x^2 \equiv -y$.

Example

Let's find the square root of 5 mod 11. Since $(p+1)/4=3$, we compute $x \equiv 5^3 \equiv 4 \pmod{11}$. Since $4^2 \equiv 5 \pmod{11}$, the square roots of 5 mod 11 are

±4.

Now let's try to find a square root of 2 mod 11. Since (p+1)/4=3, we compute $23 \equiv 8 \pmod{11}$. But $82 \equiv 9 \equiv -2 \pmod{11}$, so we have found a square root of −2 rather than of 2. This is because 2 has no square root mod 11.

We now consider square roots for a composite modulus. Note that

$$x2 \equiv 71 \pmod{77}$$

means that

$$x2 \equiv 71 \equiv 1 \pmod{7} \text{ and } x2 \equiv 71 \equiv 5 \pmod{11}.$$

Therefore,

$$x \equiv \pm 1 \pmod{7} \text{ and } x \equiv \pm 4 \pmod{11}.$$

The Chinese remainder theorem tells us

that a congruence mod 7 and a congruence mod 11 can be recombined into a congruence mod 77. For example, if $x \equiv 1 \pmod 7$ and $x \equiv 4 \pmod{11}$, then $x \equiv 15 \pmod{77}$. In this way, we can recombine in four ways to get the solutions

$x \equiv \pm 15, \pm 29 \pmod{77}$.

Now let's turn things around. Suppose $n=pq$ is the product of two primes and we know the four solutions $x \equiv \pm a, \pm b$ of $x2 \equiv y \pmod n$. From the construction just used above, we know that $a \equiv b \pmod p$ and $a \equiv -b \pmod q$ (or the same congruences with p and q switched). Therefore, $p|(a-b)$ but $q\nmid(a-b)$. This means that $\gcd(a-b, n)=p$, so we have found a nontrivial factor of n (this is

essentially the Basic Factorization Principle of Section 9.4).

For example, in the preceding example we know that $152 \equiv 292 \equiv 71 \pmod{77}$. Therefore, gcd(15−29, 77)=7 gives a nontrivial factor of 77.

Another example of computing square roots mod n is given in Section 18.1.

Notice that all the operations used above are fast, with the exception of factoring n. In particular, the Chinese remainder theorem calculation can be done quickly. So can the computation of the gcd. The modular exponentiations needed to compute square roots mod p and mod q can be done quickly using successive squaring. Therefore, we can

state the following principle:

Suppose n=pq is the product of two primes congruent to 3 mod 4, and suppose y is a number relatively prime to n that has a square root mod n. Then finding the four solutions $x \equiv \pm a, \pm b$ *to* $x2 \equiv y \pmod{n}$ *is computationally equivalent to factoring n.*

In other words, if we can find the solutions, then we can easily factor n; conversely, if we can factor n, we can easily find the solutions. For more on this, see Section 9.4.

Now suppose someone has a machine that can find single square roots mod n. That is, if we give the machine a number y that has a square root mod n,

then the machine returns one solution of $x2 \equiv y \pmod{n}$. We can use this machine to factor n as follows: Choose a random integer $x1 \pmod{n}$, compute $y \equiv x12 \pmod{n}$, and give the machine y. The machine returns x with $x2 \equiv y \pmod{n}$. If our choice of x1 is truly random, then the machine has no way of knowing the value of x1, hence it does not know whether $x \equiv x1 \pmod{n}$ or not, even if it knows all four square roots of y. So half of the time, $x \equiv \pm x1 \pmod{n}$, but half of the time, $x \equiv \pm x1 \pmod{n}$. In the latter case, we compute $\gcd(x-x1, n)$ and obtain a nontrivial factor of n. Since there is a 50% chance of success for each time we choose x1, if we choose several random values of x1, then it is very likely that we will eventually factor

n. Therefore, we conclude that any machine that can find single square roots mod n can be used, with high probability, to factor n.

3.10 Legendre and Jacobi Symbols

Suppose we want to determine whether or not $x2 \equiv a \pmod{p}$ has a solution, where p is prime. If p is small, we could square all of the numbers mod p and see if a is on the list. When p is large, this is impractical. If $p \equiv 3 \pmod{4}$, we can use the technique of the previous section and compute $s \equiv a(p+1)/4 \pmod{p}$. If a has a square root, then s is one of them, so we simply have to square s and see if we get a. If not, then a has no square root mod p. The following proposition gives a method for deciding whether a is a square mod p that works for arbitrary odd p.

Proposition

Let p be an odd prime and let a be an integer with $a \equiv 0 \pmod{p}$. Then $a^{(p-1)/2} \equiv \pm 1 \pmod{p}$. The congruence $x^2 \equiv a \pmod{p}$ has a solution if and only if $a^{(p-1)/2} \equiv 1 \pmod{p}$.

Proof. Let $y \equiv a^{(p-1)/2} \pmod{p}$. Then $y^2 \equiv a^{p-1} \equiv 1 \pmod{p}$, by Fermat's theorem. Therefore (Exercise 15), $y \equiv \pm 1 \pmod{p}$.

If $a \equiv x^2$, then $a^{(p-1)/2} \equiv x^{p-1} \equiv 1 \pmod{p}$. The hard part is showing the converse. Let α be a primitive root mod p. Then $a \equiv \alpha^j$ for some j. If $a^{(p-1)/2} \equiv 1 \pmod{p}$, then

$$\alpha^{j(p-1)/2} \equiv a^{(p-1)/2} \equiv 1 \pmod{p}.$$

By the Proposition of Section 3.7, $j(p-1)/2 \equiv 0 \pmod{p-1}$. This implies that j must be even: $j=2k$. Therefore, $a \equiv g^j \equiv (\alpha^k)^2 \pmod{p}$, so a is a square mod p.

The criterion is very easy to implement on a computer, but it can be rather difficult to use by hand. In the following, we introduce the Legendre and Jacobi symbols, which give us an easy way to determine whether or not a number is a square mod p. They also are useful in primality testing (see Section 9.3).

Let p be an odd prime and let $a \equiv 0 \pmod{p}$. Define the **Legendre symbol**

$$\left(\frac{a}{p}\right) = \begin{cases} +1 & \text{if } x^2 \equiv a \pmod{p} \text{ has a solution.} \\ -1 & \text{if } x^2 \equiv a \pmod{p} \text{ has no solution.} \end{cases}$$

Some important properties of the Legendre symbol are given in the following.

Proposition

Let p be an odd prime.

1. If $a \equiv b \equiv 0 \pmod{p}$, then

   $ap = bp$.

2. If $a \equiv 0 \pmod{p}$, then

   $(ap) \equiv a(p-1)/2 \pmod{p}$.

3. If $ab \equiv 0 \pmod{p}$, then

   $abp = apbp$.

4. $-1p = (-1)(p-1)/2$.

Proof. Part (1) is true because the solutions to $X2 \equiv a$ are the same as those to $X2 \equiv b$ when $a \equiv b \pmod{p}$.

Part (2) is the definition of the Legendre symbol combined with the previous proposition.

To prove part (3), we use part (2):

$$(abp) \equiv (ab)^{(p-1)/2} \equiv a^{(p-1)/2} b^{(p-1)/2} \equiv (ap)(bp) \pmod{p}.$$

Since the left and right ends of this congruence are ±1 and they are congruent mod the odd prime p, they must be equal. This proves (3).

For part (4), use part (2) with a=−1:

$$(-1p) \equiv (-1)^{(p-1)/2} \pmod{p}.$$

Again, since the left and right sides of this congruence are ±1 and they are congruent mod the odd prime p, they must be equal. This proves (4).

Example

Let p=11. The nonzero squares mod 11 are 1, 3, 4, 5, 9. We have

$$\left(\frac{6}{11}\right)\left(\frac{17}{11}\right)=(-1)(-1)=+1$$

and (use property (1))

$$\left(\frac{4}{11}\right)\left(\frac{21}{11}\right)=\left(\frac{9}{11}\right)=+1.$$

Therefore,

$$\left(\frac{6}{11}\right)\left(\frac{17}{11}\right)=\left(\frac{4}{11}\right)\left(\frac{21}{11}\right).$$

The Jacobi symbol extends the Legendre symbol from primes p to composite odd integers n. One might be tempted to define the symbol to be +1 if a is a square mod n and −1 if not. However, this would cause the important property (3) to fail. For example, 2 is not a square mod 35, and

3 is not a square mod 35 (since they are not squares mod 5), but also the product 6 is not a square mod 35 (since it is not a square mod 7). If Property (3) held, then we would have $(-1)(-1)=-1$, which is false.

In order to preserve property (3), we define the **Jacobi symbol** as follows. Let n be an odd positive integer and let a be a nonzero integer with gcd(a, n)=1. Let

$$n=p_1^{b_1}p_2^{b_2}\cdots p_r^{b_r}$$

be the prime factorization of n. Then

$$\left(\frac{a}{n}\right)=\left(\frac{a}{p_1}\right)^{b_1}\left(\frac{a}{p_2}\right)^{b_2}\cdots\left(\frac{a}{p_r}\right)^{b_r}.$$

The symbols on the right side are the Legendre symbols introduced earlier. Note that if n=p, the right side is simply

one Legendre symbol, so the Jacobi symbol reduces to the Legendre symbol.

Example

Let n=135=33·5. Then

2135=23325=(−1)3(−1)=+1.

Note that 2 is not a square mod 5, hence is not a square mod 135. Therefore, the fact that the Jacobi symbol has the value +1 does not imply that 2 is a square mod 135.

The main properties of the Jacobi symbol are given in the following theorem. Parts (1), (2), and (3) can be deduced from those of the Legendre symbol. Parts (4) and (5) are much deeper.

Theorem

Let n be odd.

1. If $a \equiv b \pmod{n}$ and $\gcd(a, n)=1$, then

   $(an)=(bn)$.

2. If $\gcd(ab, n)=1$, then

   $(abn)=(an)(bn)$.

3. $-1n=(-1)(n-1)/2$.

4. $(2n)=\begin{cases} +1 \text{ if } n \equiv 1 \text{ or } 7 \pmod{8} \\ -1 \text{ if } n \equiv 3 \text{ or } 5 \pmod{8}. \end{cases}$

5. Let m be odd with $\gcd(m, n)=1$. Then

   $(mn)=\begin{cases} -(nm) & \text{if } m \equiv n \equiv 3 \pmod{4} \\ +(nm) & \text{otherwise.} \end{cases}$

Note that we did not include a statement that $a^n \equiv a^{(n-1)/2}$. This is usually not true for composite n (see Exercise 45). In fact, the Solovay-Strassen primality test (see Section 9.3) is based on this

fact.

Part (5) is the famous **law of quadratic reciprocity**, proved by Gauss in 1796. When m and n are primes, it relates the question of whether m is a square mod n to the question of whether n is a square mod m.

A proof of the theorem when m and n are primes can be found in most elementary number theory texts. The extension to composite m and n can be deduced fairly easily from this case. See [Niven et al.], [Rosen], or [KraftW], for example.

When quadratic reciprocity is combined with the other properties of the Jacobi symbol, we obtain a fast way to evaluate

the symbol. Here are two examples.

Example

Let’s calculate 456712345:

$$456712345=+123454567 \quad (\text{by } (5), \text{ since } 12345 \equiv 1 \pmod 4)=$$
$$+32114567 \quad (\text{by } (1), \text{ since } 12345 \equiv 3211 \pmod{4567})=$$
$$-45673211 \quad (\text{by } (5)) \quad = \quad -13563211 \quad (\text{by } (1))=$$
$$-2321123393211 \quad (\text{by } (2), \text{ since } 1356=22\cdot 339)=-3393211$$
$$(\text{since } (\pm 1)2=1)=+3211339 \quad (\text{by } (5)) \quad = \quad +160339 \quad (\text{by } (1))=$$
$$+233955339 \quad (\text{by } (2), \text{ since } 160=25\cdot 5)=+(-1)55339 \quad (\text{by } (4))$$
$$= \quad -3395 \quad (\text{by } (5))=-45 \quad (\text{by } (1)) \quad = \quad -252=-1.$$

The only factorization needed in the calculation was removing powers of 2, which is easy to do. The fact that the calculations can be done without factoring odd numbers is important in the applications. The fact that the answer is -1 implies that 4567 is not a square mod 12345. However, if the answer had been $+1$, we could not have deduced whether 4567 is a square or is

not a square mod 12345. See Exercise 44.

Example

Let's calculate 107137:

$$107137=+137107(\text{by }(5))=+30107(\text{by }(1))$$
$$=+(2107)(15107)\quad(\text{by }(2))=+(-1)(15107)\quad(\text{by }(4))=+(10715)$$
$$(\text{by }(5))=+(215)\quad(\text{by }(1))=+1\quad(\text{by }(5)).$$

Since 137 is a prime, this says that 107 is a square mod 137. In contrast, during the calculation, we used the fact that $215=+1$. This does not mean that 2 is a square mod 15. In fact, 2 is not a square mod 5, so it cannot be a square mod 15. Therefore, although we can interpret the final answer as saying that 107 is a square mod the prime 137, we should not interpret intermediate steps involving composite numbers as saying

that a number is a square.

Suppose n=pq is the product of two large primes. If an=−1, then we can conclude that a is not a square mod n. What can we conclude if an=+1? Since

(an)=(ap)(aq),

there are two possibilities:

ap=aq=−1orap=aq=+1.

In the first case, a is not a square mod p, therefore cannot be a square mod pq.

In the second case, a is a square mod p and mod q. The Chinese remainder theorem can be used to combine a square root mod p and a square root mod q to get a square root of a mod n. Therefore, a is a square mod n.

Therefore, if an=+1, then a can be either a square or a nonsquare mod n. Deciding which case holds is called the **quadratic residuosity problem**. No fast algorithm is known for solving it. Of course, if we can factor n, then the problem can easily be solved by computing ap.

3.11 Finite Fields

Note: This section is more advanced than the rest of the chapter. It is included because finite fields are often used in cryptography. In particular, finite fields appear in four places in this book. The finite field GF(28) is used in AES (Chapter 8). Finite fields give an explanation of some phenomena that are mentioned in Section 5.2. Finally, finite fields are used in Section 21.4, Chapter 22 and in error correcting codes (Chapter 24).

Many times throughout this book, we work with the integers mod p, where p is a prime. We can add, subtract, and multiply, but what distinguishes working mod p from working mod an arbitrary integer n is that we can divide by any number that is nonzero mod p. For example, if we need to solve $3x \equiv 1 \pmod 5$, then we divide by 3 to obtain $x \equiv 2 \pmod 5$. In contrast, if we want to

solve $3x \equiv 1 \pmod 6$, there is no solution since we cannot divide by 3 $\pmod 6$. Loosely speaking, a set that has the operations of addition, multiplication, subtraction, and division by nonzero elements is called a field. We also require that the associative, commutative, and distributive laws hold.

Example

The basic examples of fields are the real numbers, the complex numbers, the rational numbers, and the integers mod a prime. The set of all integers is not a field since we sometimes cannot divide and obtain an answer in the set (for example, 4/3 is not an integer).

Example

Here is a field with four elements. Consider the set

$GF(4)=\{0, 1, \omega, \omega 2\}$,

with the following laws:

1. $0+x=x$ for all x.
2. $x+x=0$ for all x.
3. $1\cdot x=x$ for all x.
4. $\omega+1=\omega 2$.
5. Addition and multiplication are commutative and associative, and the distributive law $x(y+z)=xy+xz$ holds for all x, y, z.

Since

$\omega 3=\omega\cdot\omega 2=\omega\cdot(1+\omega)=\omega+\omega 2=\omega+(1+\omega)=1$,

we see that $\omega 2$ is the multiplicative inverse of ω. Therefore, every nonzero element of GF(4) has a multiplicative

inverse, and GF(4) is a field with four elements.

In general, a **field** is a set containing elements 0 and 1 (with $1 \neq 0$) and satisfying the following:

1. It has a multiplication and addition satisfying (1), (3), (5) in the preceding list.

2. Every element has an additive inverse (for each x, this means there exists an element $-x$ such that $x+(-x)=0$).

3. Every nonzero element has a multiplicative inverse.

A field is closed under subtraction. To compute $x-y$, simply compute $x+(-y)$.

The set of 2×2 matrices with real entries is not a field for two reasons. First, the multiplication is not commutative.

Second, there are nonzero matrices that do not have inverses (and therefore we cannot divide by them). The set of nonnegative real numbers is not a field. We can add, multiply, and divide, but sometimes when we subtract the answer is not in the set.

For every power pn of a prime, there is exactly one finite field with pn elements, and these are the only finite fields. We'll soon show how to construct them, but first let's point out that if $n>1$, then the integers mod pn do not form a field. The congruence $px \equiv 1 \pmod{pn}$ does not have a solution, so we cannot divide by p, even though $p \equiv 0 \pmod{pn}$. Therefore, we need more complicated constructions to produce fields with pn

elements.

The field with pn elements is called GF(pn). The “GF” is for “Galois field,” named for the French mathematician Evariste Galois (1811–1832), who did some early work related to fields.

Example, continued

Here is another way to produce the field GF(4). Let Z2[X] be the set of polynomials whose coefficients are integers mod 2. For example, 1+X3+X6 and X are in this set. Also, the constant polynomials 0 and 1 are in Z2[X]. We can add, subtract, and multiply in this set, as long as we work with the coefficients mod 2. For example,

(X3+X+1)(X+1)=X4+X3+X2+1

since the term 2X disappears mod 2. The important property for our purposes is that we can perform division with remainder, just as with the integers. For example, suppose we divide X2+X+1 into X4+X3+1. We can do this by long division, just as with numbers:

```
          X2+1X2+X+1X4+X3+1      X4+X3+X2  _
X2+1        X2+X+1_               X
```

In words, what we did was to divide by X2+X+1 and obtain the X2 as the first term of the quotient. Then we multiplied this X2 times X2+X+1 to get X4+X3+X2, which we subtracted from X4+X3+1, leaving X2+1. We divided this X2+1 by X2+X+1 and obtained the second term of the quotient, namely 1. Multiplying 1 times X2+X+1 and subtracting from X2+1 left the

remainder X. Since the degree of the polynomial X is less than the degree of X2+X+1, we stopped. The quotient was X2+1 and the remainder was X:

$$X4+X3+1=(X2+1)(X2+X+1)+X.$$

We can write this as

$$X4+X3+1 \equiv X(\text{mod } X2+X+1).$$

Whenever we divide by X2+X+1 we can obtain a remainder that is either 0 or a polynomial of degree at most 1 (if the remainder had degree 2 or more, we could continue dividing). Therefore, we define Z2[X] (mod X2+X+1) to be the set

$$\{0, 1, X, X+1\}$$

of polynomials of degree at most 1, since these are the remainders that we

obtain when we divide by X2+X+1. Addition, subtraction, and multiplication are done mod X2+X+1. This is completely analogous to what happens when we work with integers mod n. In the present situation, we say that two polynomials f(X) and g(X) are congruent mod X2+X+1, written $f(X) \equiv g(X) \pmod{X2+X+1}$, if f(X) and g(X) have the same remainder when divided by X2+X+1. Another way of saying this is that f(X)−g(X) is a multiple of X2+X+1. This means that there is a polynomial h(X) such that f(X)−g(X)=(X2+X+1)h(X).

Now let's multiply in $Z2[X] \pmod{X2+X+1}$. For example,

$X \cdot X = X2 \equiv X+1 \pmod{X2+X+1}$.

(It might seem that the right side should be −X−1, but recall that we are working with coefficients mod 2, so +1 and −1 are the same.) As another example, we have

$$X3 \equiv X \cdot X2 \equiv X \cdot (X+1) \equiv X2+X \equiv 1 (\mathrm{mod}\, X2+X+1).$$

It is easy to see that we are working with the set GF(4) from before, with X in place of ω.

Working with Z2[X] mod a polynomial can be used to produce finite fields. But we cannot work mod an arbitrary polynomial. The polynomial must be irreducible, which means that it doesn't factor into polynomials of lower degree mod 2. For example, X2+1, which is irreducible when we are working with real numbers, is not irreducible when

the coefficients are taken mod 2 since X2+1=(X+1)(X+1) when we are working mod 2. However, X2+X+1 is irreducible: Suppose it factors mod 2 into polynomials of lower degree. The only possible factors mod 2 are X and X +1, and X2+X+1 is not a multiple of either of these, even mod 2.

Here is the general procedure for constructing a finite field with pn elements, where p is prime and n ≥ 1. We let Zp denote the integers mod p.

1. Zp[X] is the set of polynomials with coefficients mod p.

2. Choose P(X) to be an irreducible polynomial mod p of degree n.

3. Let GF(pn) be Zp[X] mod P(X). Then GF(pn) is a field with pn elements.

The fact that GF(pn) has pn elements is easy to see. The possible remainders after dividing by P(X) are the polynomials of the form a0+a1X+⋯+an−1Xn−1, where the coefficients are integers mod p. There are p choices for each coefficient, hence pn possible remainders.

For each n, there are irreducible polynomials mod p of degree n, so this construction produces fields with pn elements for each n ≥ 1. What happens if we do the same construction for two different polynomials P1(X) and P2(X), both of degree n? We obtain two fields, call them GF(pn)′ and GF(pn)″. It is possible to show that these are essentially the same field (the technical

term is that the two fields are isomorphic), though this is not obvious since multiplication mod P1(X) is not the same as multiplication mod P2(X).

3.11.1 Division

We can easily add, subtract, and multiply polynomials in Zp[X], but division is a little more subtle. Let's look at an example. The polynomial X8+X4+X3+X+1 is irreducible in Z2[X] (although there are faster methods, one way to show it is irreducible is to divide it by all polynomials of smaller degree in Z2[X]). Consider the field

GF(28)=Z2[X](mod X8+X4+X3+X+1).

Since X7+X6+X3+X+1 is not 0, it

should have an inverse. The inverse is found using the analog of the extended Euclidean algorithm. First, perform the gcd calculation for gcd(X7+X6+X3+X+1, X8+X4+X3+X+1). The procedure (remainder → divisor → dividend → ignore) is the same as for integers:

X8+X4+X3+X+1=(X+1)(X7+X6+X3+X+1)+(X6+X2+X)X7+X6+X3+X+1=(X+1)(X6+X2+X)+1.

The last remainder is 1, which tells us that the “greatest common divisor” of X7+X6+X3+X+1 and X8+X4+X3+X+1 is 1. Of course, this must be the case, since X8+X4+X3+X+1 is irreducible, so its only factors are 1 and itself.

Now work the Extended Euclidean algorithm to express 1 as a linear combination of X7+X6+X3+X+1 and

X8+X4+X3+X+1:

The end result is

$$1=(X2)(X7+X6+X3+X+1)+(X+1)(X8+X4+X3+X+1).$$

Reducing mod X8+X4+X3+X+1, we obtain

$$(X2)(X7+X6+X3+X+1) \equiv 1 (\text{mod } X8+X4+X3+X+1),$$

which means that X2 is the multiplicative inverse of X7+X6+X3+X+1. Whenever we need to divide by X7+X6+X3+X+1, we can instead multiply by X2. This is the analog of what we did when working with the usual integers mod p.

3.11.2 *GF*(2^8)

In Chapter 8, we discuss AES, which uses GF(28), so let's look at this field a little more closely. We'll work mod the irreducible polynomial X8+X4+X3+X+1, since that is the one used by AES.

However, there are other irreducible polynomials of degree 8, and any one of them would lead to similar calculations. Every element can be represented uniquely as a polynomial

b7X7+b6X6+b5X5+b4X4+b3X3+b2X2+b1X+b0,

where each bi is 0 or 1. The 8 bits b7b6b5b4b3b2b1b0 represent a byte, so we can represent the elements of GF(28) as 8-bit bytes. For example, the polynomial X7+X6+X3+X+1 becomes 11001011. Addition is the XOR of the bits:

(X7+X6+X3+X+1)+(X4+X3+1)
→11001011⊕00011001=11010010 →X7+X6+X4+X.

Multiplication is more subtle and does not have as easy an interpretation. That is because we are working mod the

polynomial X8+X4+X3+X+1, which we can represent by the 9 bits 100011011. First, let's multiply X7+X6+X3+X+1 by X: With polynomials, we calculate

(X7+X6+X3+X+1)
(X)=X8+X7+X4+X2+X=(X7+X3+X2+1)+(X8+X4+X3+X
+1) ≡ X7+X3+X2+1(mod X8+X4+X3+X+1).

The same operation with bits becomes

11001011→110010110(shift left and append a 0)→110010110⊕100011011(su
+X4+X3+X+1)=010001101,

which corresponds to the preceding answer. In general, we can multiply by X by the following algorithm:

1. Shift left and append a 0 as the last bit.
2. If the first bit is 0, stop.
3. If the first bit is 1, *XOR* with 100011011.

The reason we stop in step 2 is that if the first bit is 0 then the polynomial still has degree less than 8 after we multiply by X, so it does not need to be reduced. To multiply by higher powers of X, multiply by X several times. For example, multiplication by X3 can be done with three shifts and at most three *XOR*s. Multiplication by an arbitrary polynomial can be accomplished by multiplying by the various powers of X appearing in that polynomial, then adding (i.e., *XOR*ing) the results.

In summary, we see that the field operations of addition and multiplication in GF(28) can be carried out very efficiently. Similar considerations apply to any finite field.

The analogy between the integers mod a prime and polynomials mod an irreducible polynomial is quite remarkable. We summarize in the following.

integers↔Zp[X]prime number q↔irreducible P(X) of degree nZq↔Zp[X] (mod P(X))field with q elements↔field with pn elements

Let GF(pn)∗ denote the nonzero elements of GF(pn). This set, which has pn−1 elements, is closed under multiplication, just as the integers not congruent to 0 mod p are closed under multiplication. It can be shown that there is a generating polynomial g(X) such that every element in GF(pn)∗ can be expressed as a power of g(X). This also means that the smallest exponent k such that g(X)k ≡ 1 is pn−1. This is the

analog of a primitive root for primes. There are $\phi(p^n-1)$ such generating polynomials, where ϕ is Euler's function. An interesting situation occurs when $p=2$ and $2n-1$ is prime. In this case, every nonzero polynomial $f(X) \neq 1$ in $GF(2n)$ is a generating polynomial. (*Remark, for those who know some group theory:* The set $GF(2n)^*$ is a group of prime order in this case, so every element except the identity is a generator.)

The **discrete log problem** mod a prime, which we'll discuss in Chapter 10, has an analog for finite fields; namely, given $h(x)$, find an integer k such that $h(X)=g(X)^k$ in $GF(p^n)$. Finding such a k is believed to be very hard in most

situations.

3.11.3 LFSR Sequences

We can now explain a phenomenon that is mentioned in Section 5.2 on LFSR sequences.

Suppose that we have a recurrence relation

$$x_{n+m} \equiv c_0 x_n + c_1 x_{n+1} + \cdots + c_{m-1} x_{n+m-1} \pmod 2.$$

For simplicity, we assume that the associated polynomial

$$P(X) = X^m + c_{m-1} X^{m-1} + c_{m-2} X^{m-2} + \cdots + c_0$$

is irreducible mod 2. Then $Z_2[X] \pmod{P(X)}$ is the field $GF(2^m)$. We regard $GF(2^m)$ as a vector space over Z_2 with basis $\{1, X, X^2, X^3, \ldots, X^{m-1}\}$.

Multiplication by X gives a linear transformation of this vector space. Since

$$X\cdot 1=X,\ X\cdot X=X2,\ X\cdot X2=X3,\ \ldots$$
$$X\cdot Xm-1=Xm\equiv c0+c1X+\cdots+cm-1Xm-1,$$

multiplication by X is represented by the matrix

$$MX=00\cdots0c010\cdots0c101\cdots0c2\vdots\ddots\vdots0001cm-1.$$

Suppose we know (xn, xn+1, xn+2, … , xn+m−1). We compute

$$(xn,\ xn+1,\ xn+2,\ \ldots,\ xn+m-1)MX=(xn+1,\ xn+2,\ xn+3,$$
$$\ldots,\ c0xn+\cdots+cm-1xn+m-1)\equiv(xn+1,\ xn+2,\ xn+3,\ \ldots,\ xn+m).$$

Therefore, multiplication by MX shifts the indices by 1. It follows easily that multiplication on the right by the matrix MXj sends (x1, x2, … , xm) to (x1+j, x2+j, … , xm+j). If MXj ≡ I, the identity matrix, this must be the original

vector (x1, x2, … , xm). Since there are 2m−1 nonzero elements in GF(2m), it follows from Lagrange's theorem in group theory that $X^{2m-1} \equiv 1$, which implies that $MX^{2m-1}=I$. Therefore, we know that $x1 \equiv x2m$, $x2 \equiv x2m+1$, ….

For any set of initial values (we'll assume that at least one initial value is nonzero), the sequence will repeat after k terms, where k is the smallest positive integer such that $X^k \equiv 1 \pmod{P(X)}$. It can be shown that k divides 2m−1.

In fact, the period of such a sequence is exactly k. This can be proved as follows, using a few results from linear algebra: Let $v=(x1, \ldots, xm) \neq 0$ be the row vector of initial values. The sequence repeats when $vMX^j=v$. This

means that the nonzero *row* vector v is in the left null space of the matrix MXj −I, so det(MXj−I)=0. But this means that there is a nonzero *column* vector w=(a0, … , am−1)T in the right null space of MXj−I. That is, MXjw=w. Since the matrix MXj represents the linear transformation given by multiplication by Xj with respect to the basis {1, X, … , Xm−1}, this can be changed back into a relation among polynomials:

$$Xj(a0+a1X+\cdots+am-1Xm-1) \equiv a0+a1X+\cdots+am-1Xm-1 \pmod{P(X)}.$$

But $a0+a1X+\cdots+am-1Xm-1 \pmod{P(X)}$ is a nonzero element of the field GF(2m), so we can divide by this element to get $Xj \equiv 1 \pmod{P(X)}$. Since j=k is the first time this happens, the

sequence first repeats after k terms, so it has period k.

As mentioned previously, when $2m-1$ is prime, all polynomials (except 0 and 1) are generating polynomials for GF(2m). In particular, X is a generating polynomial and therefore $k=2m-1$ is the period of the recurrence.

3.12 Continued Fractions

There are many situations where we want to approximate a real number by a rational number. For example, we can approximate π=3.14159265… by 314/100=157/50. But 22/7 is a slightly better approximation, and it is more efficient in the sense that it uses a smaller denominator than 157/50. The method of continued fractions is a procedure that yields this type of good approximations. In this section, we summarize some basic facts. For proofs and more details, see, for example, [Hardy-Wright], [Niven et al.], [Rosen], and [KraftW].

An easy way to approximate a real number x is to take the largest integer less than or equal to x. This is often denoted by [x]. For example, [π]=3. If we want to get a better approximation, we need to look at the remaining fractional part. For π=3.14159…, this is .14159…. This looks close to 1/7=.142857…. One way to express this is to look at 1/.14159=7.06251. We can approximate this last number by [7.06251…]=7 and therefore conclude that 1/7 is indeed a good approximation for .14159 and that 22/7 is a good approximation for π. Continuing in this manner yields even better approximations. For example, the next step is to compute 1/.06251=15.9966 and then take the greatest integer to get

15 (yes, 16 is closer, but the algorithm corrects for this in the next step). We now have

$$\pi \approx 3+17+115=333106.$$

If we continue one more step, we obtain

$$\pi=3+17+115+11=355113.$$

This last approximation is very accurate:

$\pi=3.14159265\ldots,$ and $355/113=3.14159292\ldots.$

This procedure works for arbitrary real numbers. Start with a real number x. Let a0=[x] and x0=x. Then (if xi ≠ ai; otherwise, stop) define

$$xi+1=1xi-ai,\ ai+1=[xi+1].$$

We obtain the approximations

$$pnqn=a0+1a1+1a2+1\cdots+1an.$$

We have therefore produced a sequence of rational numbers p1/q1, p2/q2, …. It can be shown that each rational number pk/qk gives a better approximation to x than any of the preceding rational numbers pj/qj with $1 \le j<k$. Moreover, the following holds.

Theorem

If $|x-(r/s)| < 1/2s2$ for integers r, s, then r/s=pi/qi for some i.

For example, $|\pi-22/7| \approx .001<1/98$ and 22/7=p2/q2.

Continued fractions yield a convenient way to recognize rational numbers from their decimal expansions. For example, suppose we encounter the decimal

3.764705882 and we suspect that it is the beginning of the decimal expansion of a rational number with small denominator. The first few terms of the continued fraction are

$$3+\frac{1}{1+\frac{1}{3+\frac{1}{1+\frac{1}{4+\frac{1}{9803921}}}}}.$$

The fact that 9803921 is large indicates that the preceding approximation is quite good, so we calculate

$$3+\frac{1}{1+\frac{1}{3+\frac{1}{1+\frac{1}{4}}}}=\frac{64}{17}=3.7647058823529\ldots,$$

which agrees with all of the terms of the original 3.764605882. Therefore, 64/17 is a likely candidate for the answer. Note that if we had included the 9803921, we would have obtained a fraction that also agrees with the original decimal expansion but has a significantly larger denominator.

Now let’s apply the procedure to 12345/11111. We have

$$12345/11111 = 1 + 1/(9 + 1/(246 + 1/(1 + 1/4))).$$

This yields the numbers

$$1,\ 10/9,\ 2461/2215,\ 2471/2224,\ 12345/11111.$$

Note that the numbers 1, 9, 246, 1, 4 are the quotients obtained during the computation of gcd(12345, 11111) in Subsection 3.1.3 (see Exercise 49).

Calculating the fractions such as

$$2461/2215 = 1 + 1/(9 + 1/246)$$

can become tiresome when done in the straightforward way. Fortunately, there is a faster method. Define

$$p_{-2} = 0,\ p_{-1} = 1,\ q_{-2} = 1,\ q_{-1} = 0,\ p_{n+1} = a_{n+1}p_n + p_{n-1}\ q_{n+1} = a_{n+1}q_n + q_{n-1}.$$

Then

$$\frac{p_n}{q_n} = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{\cdots + \frac{1}{a_n}}}}.$$

Using these relations, we can compute the partial quotients p_n/q_n from the previous ones, rather than having to start a new computation every time a new a_n is found.

3.13 Exercises

1.

    1. Find integers x and y such that 17x +101y=1.

    2. Find 17−1 (mod 101).

2.

    1. Using the identity x3+y3=(x+y)(x2−xy +y3), factor 2333+1 into a product of two integers greater than 1.

    2. Using the congruence 22 ≡ 1 (mod 3), deduce that 2232 ≡ 1 (mod 3) and show that 2333+1 is a multiple of 3.

3.

    1. Solve 7d ≡ 1 (mod 30).

    2. Suppose you write a message as a number m (mod 31). Encrypt m as m7 (mod 31). How would you decrypt? (Hint: Decryption is done by raising the ciphertext to a power mod 31. Fermat's theorem will be useful.)

4. Solve $5x+2 \equiv 3x-7 \pmod{31}$.

5.

1. Find all solutions of $12x \equiv 28 \pmod{236}$.

2. Find all solutions of $12x \equiv 30 \pmod{236}$.

6.

1. Find all solutions of $4x \equiv 20 \pmod{50}$.

2. Find all solutions of $4x \equiv 21 \pmod{50}$.

7.

1. Let $n \geq 2$. Show that if n is composite then n has a prime factor $p \leq n$.

2. Use the Euclidean algorithm to compute gcd(30030, 257).

3. Using the result of parts (a) and (b) and the fact that $30030=2\cdot3\cdot5\cdot7\cdot11\cdot13$, show that 257 is prime. (Remark: This method of computing one gcd, rather than doing several trial divisions (by 2, 3, 5, ...), is often faster for checking whether small primes divide a number.)

8. Compute gcd(12345678987654321, 100).

9.

1. Compute gcd(4883, 4369).

2. Factor 4883 and 4369 into products of primes.

10.

1. What is gcd(111, 11)? Using the Extended Euclidean algorithm, find $11^{-1} \bmod 111$.

2. What is gcd(1111, 11)? Does $11^{-1} \bmod 1111$ exist?

3. Find gcd(x, 11), where x consists of n repeated 1s. What can you say about $11^{-1} \bmod x$ as a function of n?

11.

1. Let $F1=1$, $F2=1$, $Fn+1=Fn+Fn-1$ define the Fibonacci numbers 1, 1, 2, 3, 5, 8, …. Use the Euclidean algorithm to compute gcd(Fn, Fn−1) for all $n \geq 1$.

2. Find gcd(11111111, 11111).

3. Let $a=111\cdots11$ be formed with Fn repeated 1's and let $b=111\cdots11$ be formed with Fn−1 repeated 1's. Find gcd(a, b). (Hint: Compare your computations in parts (a) and (b).)

12. Let $n \geq 2$. Show that none of the numbers $n!+2$, $n!+3$, $n!+4$, … , $n!+n$ are prime.

13.

    1. Let p be prime. Suppose a and b are integers such that $ab \equiv 0 \pmod{p}$. Show that either $a \equiv 0$ or $b \equiv 0 \pmod{p}$.

    2. Show that if a, b, n are integers with $n|ab$ and $\gcd(a, n)=1$, then $n|b$.

14. Let p be prime.

    1. Show that if $x^2 \equiv 0 \pmod{p}$, then $x \equiv 0 \pmod{p}$.

    2. Show that if $k \geq 2$, then $x^2 \equiv 0 \pmod{p^k}$ has solutions with $x \equiv 0 \pmod{p^k}$.

15. Let $p \geq 3$ be prime. Show that the only solutions to $x^2 \equiv 1 \pmod{p}$ are $x \equiv \pm 1 \pmod{p}$. (Hint: Apply Exercise 13(a) to $(x+1)(x-1)$.)

16. Find x with $x \equiv 3 \pmod{5}$ and $x \equiv 9 \pmod{11}$.

17. Suppose $x \equiv 2 \pmod{7}$ and $x \equiv 3 \pmod{10}$. What is x congruent to mod 70?

18. Find x with $2x \equiv 1 \pmod{7}$ and $4x \equiv 2 \pmod{9}$. (Hint: Replace $2x \equiv 1 \pmod{7}$ with $x \equiv a \pmod{7}$

for a suitable a, and similarly for the second congruence.)

19. A group of people are arranging themselves for a parade. If they line up three to a row, one person is left over. If they line up four to a row, two people are left over, and if they line up five to a row, three people are left over. What is the smallest possible number of people? What is the next smallest number? (Hint: Interpret this problem in terms of the Chinese remainder theorem.)

20. You want to find x such that when you divide x by each of the numbers from 2 to 10, the remainder is 1. The smallest such x is x=1. What is the next smallest x? (The answer is less than 3000.)

21.

    1. Find all four solutions to $x^2 \equiv 133 \pmod{143}$. (Note that $143=11\cdot13$.)

    2. Find all solutions to $x^2 \equiv 77 \pmod{143}$. (There are only two solutions in this case. This is because $\gcd(77, 143) \neq 1$.)

22. You need to compute $12345678965537 \pmod{581859289607}$. A friend offers to help: 1 cent for each multiplication mod 581859289607. Your friend is hoping to get more than $650. Describe how you can have the friend do the computation

for less than 25 cents. (Note: 65537=216+1 is the most commonly used RSA encryption exponent.)

23. Divide 210203 by 101. What is the remainder?

24. Divide 3987654321 by 11. What is the remainder?

25. Find the last 2 digits of 123562.

26. Let $p \equiv 3 \pmod 4$ be prime. Show that $x2 \equiv -1 \pmod p$ has no solutions. (Hint: Suppose x exists. Raise both sides to the power $(p-1)/2$ and use Fermat's theorem. Also, $(-1)(p-1)/2=-1$ because $(p-1)/2$ is odd.)

27. Let p be prime. Show that $ap \equiv a \pmod p$ for all a.

28. Let p be prime and let a and b be integers. Show that $(a+b)p \equiv ap+bp \pmod p$.

29.

    1. Evaluate $77 \pmod 4$.

    2. Use part (a) to find the last digit of 777. (Note: abc means a(bc) since the other possible interpretation would be (ab)c=abc, which is written more easily without a second exponentiation.) (Hint: Use part (a) and the Basic Principle that follows Euler's Theorem.)

30. You are told that exactly one of the numbers

    21000+277, 21000+291, 21000+297

    is prime and you have one minute to figure out which one. Describe calculations you could do (with software such as MATLAB or Mathematica) that would give you a very good chance of figuring out which number is prime? Do not do the calculations. Do not try to factor the numbers. They do not have any prime factors less than 109. You may use modular exponentiation, but you may not use commands of the form "IsPrime[n]" or "NextPrime[n]." (See Computer Problem 3 below.)

31.

1. Let p=7, 13, or 19. Show that $a^{1728} \equiv 1 \pmod{p}$ for all a with $p \nmid a$.

2. Let p=7, 13, or 19. Show that $a^{1729} \equiv a \pmod{p}$ for all a. (Hint: Consider the case $p \mid a$ separately.)

3. Show that $a^{1729} \equiv a \pmod{1729}$ for all a. Composite numbers n such that $a^n \equiv a \pmod{n}$ for all a are called Carmichael numbers. They are rare (561 is another example), but there are infinitely many of them [Alford et al. 2].

32.

1. Show that $210 \equiv 1 \pmod{11}$ and $25 \equiv 1 \pmod{31}$.

2. Show that $2340 \equiv 1 \pmod{341}$.

3. Is 341 prime?

33.

1. Let p be prime and let $a \equiv 0 \pmod{p}$. Let $b \equiv a^{p-2} \pmod{p}$. Show that $ab \equiv 1 \pmod{p}$.

2. Use the method of part (a) to solve $2x \equiv 1 \pmod{7}$.

34. You are appearing on the Math Superstars Show and, for the final question, you are given a 500-digit number n and are asked to guess whether or not it is prime. You are told that n is either prime or the product of a 200-digit prime and a 300-digit prime. You have one minute, and fortunately you have a computer. How would you make a guess that's very probably correct? Name any theorems that you are using.

35.

1. Compute $\phi(d)$ for all of the divisors of 10 (namely, 1, 2, 5, 10), and find the sum of these $\phi(d)$.

2. Repeat part (a) for all of the divisors of 12.

3. Let $n \geq 1$. Conjecture the value of $\sum \phi(d)$, where the sum is over the divisors of n. (This result is proved in many elementary number theory texts.)

36. Find a number α mod 7 that is a primitive root mod 7 and find a number $\gamma \equiv 0 \pmod 7$ that is not a primitive root mod 7. Show that α and γ have the desired properties.

37.

1. Show that every nonzero congruence class mod 11 is a power of 2, and therefore 2 is a primitive root mod 11.

2. Note that $23 \equiv 8 \pmod{11}$. Find x such that $8x \equiv 2 \pmod{11}$. (Hint: What is the inverse of 3 (mod 10)?)

3. Show that every nonzero congruence class mod 11 is a power of 8, and therefore 8 is a primitive root mod 11.

4. Let p be prime and let α be a primitive root mod p. Let $h \equiv \alpha y \pmod p$ with $\gcd(y, p-1)=1$. Let $xy \equiv 1 \pmod{p-1}$. Show that $hx \equiv \alpha \pmod p$.

5. Let p and h be as in part (d). Show that h is a primitive root mod p. (Remark: Since there are $\phi(p-1)$ possibilities for the exponent x in part (d), this yields all of the $\phi(p-1)$ primitive roots mod p.)

6. Use the method of part (e) to find all primitive roots for p=13, given that 2 is a primitive root.

38. It is known that 14 is a primitive root for the prime p=30000001. Let $b \equiv 149000000 \pmod{p}$. (The exponent is $3(p-1)/10$.)

    1. Explain why $b10 \equiv 1 \pmod{p}$.

    2. Explain why $b \equiv 1 \pmod{p}$.

39.

    1. Find the inverse of $1161 \pmod{26}$.

    2. Find all values of $b \pmod{26}$ such that $11b1 \pmod{26}$ is invertible.

40. Find the inverse of $1110 \pmod{2}$.

41. Find all primes p for which $3573 \pmod{p}$ is not invertible.

42.

1. Use the Legendre symbol to show that $x^2 \equiv 5 \pmod{19}$ has a solution.

2. Use the method of Section 3.9 to find a solution to $x^2 \equiv 5 \pmod{19}$.

43. Use the Legendre symbol to determine which of the following congruences have solutions (each modulus is prime):

    1. $X^2 \equiv 123 \pmod{401}$

    2. $X^2 \equiv 43 \pmod{179}$

    3. $X^2 \equiv 1093 \pmod{65537}$

44.

    1. Let n be odd and assume gcd(a, n)=1. Show that if $a^n = -1$, then a is not a square mod n.

    2. Show that $3^{35} = +1$.

    3. Show that 3 is not a square mod 35.

45. Let n=15. Show that $(2n) \equiv 2^{(n-1)/2} \pmod{n}$.

46.

    1. Show that $3^{65537} = -1$.

2. Show that $3^{(65537-1)/2} \equiv 1 \pmod{65537}$.

3. Use the procedure of Exercise 54 to show that 3 is a primitive root mod 65537. (Remark: The same proof shows that 3 is a primitive root for any prime $p \geq 5$ such that $p-1$ is a power of 2. However, there are only six known primes with $p-1$ a power of 2; namely, 2, 3, 5, 17, 257, 65537. They are called *Fermat primes*.)

47.

1. Show that the only irreducible polynomials in $Z_2[X]$ of degree at most 2 are X, $X+1$, and X^2+X+1.

2. Show that X^4+X+1 is irreducible in $Z_2[X]$. (Hint: If it factors, it must have at least one factor of degree at most 2.)

3. Show that $X^4 \equiv X+1$, $X^8 \equiv X^2+1$, and $X^{16} \equiv X \pmod{X^4+X+1}$.

4. Show that $X^{15} \equiv 1 \pmod{X^4+X+1}$.

48.

1. Show that X^2+1 is irreducible in $Z_3[X]$.

2. Find the multiplicative inverse of 1+2X in Z3[X] (mod X2+1).

49. Show that the quotients in the Euclidean algorithm for gcd(a, b) are exactly the numbers a0, a1, … that appear in the continued fraction of a/b.

50.

1. Compute several steps of the continued fractions of 3 and 7. Do you notice any patterns? (It can be shown that the ai's in the continued fraction of every irrational number of the form a+bd with a, b, d rational and d>0 eventually become periodic.)

2. For each of d=3, 7, let n be such that an +1=2a0 in the continued fraction of d. Compute pn and qn and show that x=pn and y=qn give a solution of what is known as Pell's equation: x2−dy2=1.

3. Use the method of part (b) to solve x2−19y2=1.

51. Compute several steps of the continued fraction expansion of e. Do you notice any patterns? (On the other hand, the continued fraction expansion of π seems to be fairly random.)

52. Compute several steps of the continued fraction expansion of (1+5)/2 and compute the corresponding numbers pn and qn (defined in Section 3.12). The sequences p0, p1, p2, … and q1, q2, … are what famous sequence of numbers?

53. Let a and n>1 be integers with gcd(a, n)=1. The **order** of a mod n is the smallest positive integer r such that $a^r \equiv 1 \pmod{n}$. We denote r=ordn(a).

    1. Show that $r \leq \phi(n)$.

    2. Show that if m=rk is a multiple of r, then $a^m \equiv 1 \pmod{n}$.

    3. Suppose $a^t \equiv 1 \pmod{n}$. Write t=qr+s with $0 \leq s<r$ (this is just division with remainder). Show that $a^s \equiv 1 \pmod{n}$.

    4. Using the definition of r and the fact that $0 \leq s<r$, show that s=0 and therefore r|t. This, combined with part (b), yields the result that $a^t \equiv 1 \pmod{n}$ if and only if ordn(a)|t.

    5. Show that ordn(a)|ϕ(n).

54. This exercise will show by example how to use the results of Exercise 53 to prove a number is a primitive root mod a prime p, once we know the

factorization of $p-1$. In particular, we'll show that 7 is a primitive root mod 601. Note that $600=23\cdot3\cdot52$.

1. Show that if an integer $r<600$ divides 600, then it divides at least one of 300, 200, 120 (these numbers are 600/2, 600/3, and 600/5).

2. Show that if $\text{ord}_{601}(7)<600$, then it divides one of the numbers 300, 200, 120.

3. A calculation shows that

   $7^{300}\equiv 600,\ 7^{200}\equiv 576,\ 7^{120}\equiv 423 \pmod{601}$.

   Why can we conclude that $\text{ord}_{601}(7)$ does not divide 300, 200, or 120?

4. Show that 7 is a primitive root mod 601.

5. In general, suppose p is a prime and $p-1=q_1^{a_1}\cdots q_s^{a_s}$ is the factorization of $p-1$ into primes. Describe a procedure to check whether a number α is a primitive root mod p. (Therefore, if we need to find a primitive root mod p, we can simply use this procedure to test the numbers $\alpha=2, 3, 5, 6, \ldots$ in succession until we

find one that is a primitive root.)

55. We want to find an exponent k such that $3k \equiv 2 \pmod{65537}$.

    1. Observe that $232 \equiv 1 \pmod{65537}$, but $216 \equiv 1 \pmod{65537}$. It can be shown (Exercise 46) that 3 is a primitive root mod 65537, which implies that $3n \equiv 1 \pmod{65537}$ if and only if $65536|n$. Use this to show that $2048|k$ but 4096 does not divide k. (Hint: Raise both sides of $3k \equiv 2$ to the 16th and to the 32nd powers.)

    2. Use the result of part (a) to conclude that there are only 16 possible choices for k that need to be considered. Use this information to determine k. This problem shows that if $p-1$ has a special structure, for example, a power of 2, then this can be used to avoid exhaustive searches. Therefore, such primes are cryptographically weak. See Exercise 12 in Chapter 10 for a reinterpretation of the present problem.

56.

    1. Let $x=b_1b_2 \ldots b_w$ be an integer written

in binary (for example, when x=1011, we have b1=1, b2=0, b3=1, b4=1). Let y and n be integers. Perform the following procedure:

1. Start with k=1 and s1=1.
2. If bk=1, let rk ≡ sky (mod n). If bk=0, let rk=sk.
3. Let sk+1 ≡ rk2 (mod n).
4. If k=w, stop. If k<w, add 1 to k and go to (2).

Show that rw ≡ yx (mod n).

2. Let x, y, and n be positive integers. Show that the following procedure computes yx (mod n).

1. Start with a=x, b=1, c=y.
2. If a is even, let a=a/2, and let b=b, c ≡ c2 (mod n).
3. If a is odd, let a=a−1, and let b ≡ bc (mod n), c=c.
4. If a ≠ 0, go to step 2.

5. Output b.

(Remark: This algorithm is similar to the one in part (a), but it uses the binary bits of x in reverse order.)

57. Here is how to construct the x guaranteed by the general form of the Chinese remainder theorem. Suppose $m_1, \ldots, m_k$ are integers with $\gcd(m_i, m_j)=1$ whenever $i \neq j$. Let $a_1, \ldots, a_k$ be integers. Perform the following procedure:

    1. For $i=1, \ldots, k$, let $z_i=m_1 \cdots m_{i-1} m_{i+1} \cdots m_k$.

    2. For $i=1, \ldots, k$, let $y_i \equiv z_i^{-1} \pmod{m_i}$.

    3. Let $x=a_1 y_1 z_1 + \cdots + a_k y_k z_k$.

    Show $x \equiv a_i \pmod{m_i}$ for all i.

58. Alice designs a cryptosystem as follows (this system is due to Rabin). She chooses two distinct primes p and q (preferably, both p and q are congruent to 3 mod 4) and keeps them secret. She makes $n=pq$ public. When Bob wants to send Alice a message m, he computes $x \equiv m^2 \pmod{n}$ and sends x to Alice. She makes a decryption machine that does the following: When the machine is

given a number x, it computes the square roots of x modn since it knows p and q. There is usually more than one square root. It chooses one at random, and gives it to Alice. When Alice receives x from Bob, she puts it into her machine. If the output from the machine is a meaningful message, she assumes it is the correct message. If it is not meaningful, she puts x into the machine again. She continues until she gets a meaningful message.

1. Why should Alice expect to get a meaningful message fairly soon?

2. If Oscar intercepts x (he already knows n), why should it be hard for him to determine the message m?

3. If Eve breaks into Alice's office and thereby is able to try a few chosen-ciphertext attacks on Alice's decryption machine, how can she determine the factorization of n?

59. This exercise shows that the Euclidean algorithm computes the gcd. Let a, b, qi, ri be as in Subsection 3.1.3.

    1. Let d be a common divisor of a, b. Show that d|r1, and use this to show that d|r2.

2. Let d be as in (a). Use induction to show that $d|r_i$ for all i. In particular, $d|r_k$, the last nonzero remainder.

3. Use induction to show that $r_k|r_i$ for $1 \le i \le k$.

4. Using the facts that $r_k|r_1$ and $r_k|r_2$, show that $r_k|b$ and then $r_k|a$. Therefore, r_k is a common divisor of a, b.

5. Use (b) to show that $r_k \ge d$ for all common divisors d, and therefore r_k is the greatest common divisor.

60. Let p and q be distinct primes.

1. Show that among the integers m satisfying $1 \le m < pq$, there are $q-1$ multiples of p, and there are $p-1$ multiples of q.

2. Suppose $\gcd(m, pq) > 1$. Show that m is a multiple of p or a multiple of q.

3. Show that if $1 \le m < pq$, then m cannot be a multiple of both p and q.

4. Show that the number of integers m with $1 \le m < q$ such that $\gcd(m, pq) = 1$ is pq

$-1-(p-1)-(q-1)=(p-1)(q-1)$. (Remark: This proves the formula that $\phi(pq)=(p-1)(q-1)$.)

61.

1. Give an example of integers $m \neq n$ with gcd(m, n)gt1 and integers a, b such that the simultaneous congruences

   $$x \equiv a \pmod{m},\ x \equiv b \pmod{n}$$

   have no solution.

2. Give an example of integers $m \neq n$ with gcd(m, n)>1 and integers $a \neq b$ such that the simultaneous congruences

   $$x \equiv a \pmod{m},\ x \equiv b \pmod{n}$$

   have a solution.

3.14 Computer Problems

1. Evaluate gcd(8765, 23485).

2.
    1. Find integers x and y with 65537x +3511y=1.

    2. Find integers x and y with 65537x +3511y=17.

3. You are told that exactly one of the numbers

   21000+277, 21000+291, 21000+297

   is prime and you have one minute to figure out which one. They do not have any prime factors less than 109. You may use modular exponentiation, but you may not use commands of the form “IsPrime[n]” or “NextPrime[n].” (This makes explicit Exercise 30 above.)

4. Find the last five digits of 31234567. (Note: Don’t ask the computer to print 31234567. It is too large!)

5. Look at the decimal expansion of

e=2.71828182845904523 …. Find the consecutive digits 71, the consecutive digits 271, and the consecutive digits 4523 form primes. Find the first set of five consecutive digits that form a prime (04523 does not count as a five-digit number).

6. Solve $314x \equiv 271 \pmod{11111}$.

7. Find all solutions to $216x \equiv 66 \pmod{606}$.

8. Find an integer such that when it is divided by 101 the remainder is 17, when it is divided by 201 the remainder is 18, and when it is divided by 301 the remainder is 19.

9. Let $n=391=17\cdot 23$. Show that $2^{n-1} \equiv 1 \pmod{n}$. Find an exponent $j>0$ such that $2^j \equiv 1 \pmod{n}$.

10. Let $n=84047\cdot 65497$. Find x and y with $x^2 \equiv y^2 \pmod{n}$ but $x \equiv \pm y \pmod{n}$.

11. Let $M=1241525114196$.

    1. Find the inverse of $M \pmod{101}$.

    2. For which primes p does M not have an inverse mod p?

12. Find the square roots of 26055 mod the prime 34807.

13. Find all square roots of 1522756 mod 2325781.

14. Try to find a square root of 48382 mod the prime 83987, using the method of Section 3.9. Square your answer to see if it is correct. What number did you find the square root of?

Chapter 4 The One-Time Pad

The one-time pad, which is an unbreakable cryptosystem, was described by Frank Miller in 1882 as a means of encrypting telegrams. It was rediscovered by Gilbert Vernam and Joseph Mauborgne around 1918. In terms of security, it is the best possible system, but implementation makes it unsuitable for most applications.

In this chapter, we introduce the one-time pad and show why a given key should not be used more than once. We then introduce the important concepts of perfect secrecy and ciphertext

indistinguishability, topics that have become prominent in cryptography in recent years.

4.1 Binary Numbers and ASCII

In many situations involving computers, it is more natural to represent data as strings of 0s and 1s, rather than as letters and numbers.

Numbers can be converted to binary (or base 2), if desired, which we'll quickly review. Our standard way of writing numbers is in base 10. For example, 123 means $1\times102+2\times101+3$. Binary uses 2 in place of 10 and needs only the digits 0 and 1. For example, 110101 in binary represents $25+24+22+1$ (which equals 53 in base 10).

Each 0 or 1 is called a **bit**. A representation that takes eight bits is called an eight-bit number, or a **byte**. The largest number that 8 bits can represent is 255, and the largest number that 16 bits can represent is 65535.

Often, we want to deal with more than just numbers. In this case, words, symbols, letters, and numbers are given binary representations. There are many possible ways of doing this. One of the standard ways is called ASCII, which stands for American Standard Code for Information Interchange. Each character is represented using seven bits, allowing for 128 possible characters and symbols to be represented. Eight-bit blocks are common for computers to use, and for

this reason, each character is often represented using eight bits. The eighth bit can be used for checking parity to see if an error occurred in transmission, or is often used to extend the list of characters to include symbols such as ü and è .

Table 4.1 gives the ASCII equivalents for some standard symbols.

Table 4.1 ASCII Equivalents of Selected Symbols

symbol	!	”	#	$	%	&	’
decimal	33	34	35	36	37	38	39
binary	0100001	0100010	0100011	0100100	0100101	0100110	0100111
(	)	*	+	,	-	.	/
40	41	42	43	44	45	46	47
0101000	0101001	0101010	0101011	0101100	0101101	0101110	0101111
0	1	2	3	4	5	6	7
48	49	50	51	52	53	54	55
0110000	0110001	0110010	0110011	0110100	0110101	0110110	0110111
8	9	:	;	<	=	>	?
56	57	58	59	60	61	62	63
0111000	0111001	0111010	0111011	0111100	0111101	0111110	0111111
@	A	B	C	D	E	F	G
64	65	66	67	68	69	70	71
1000000	1000001	1000010	1000011	1000100	1000101	1000110	1000111

Table 4.1 Full Alternative Text

4.2 One-Time Pads

Start by representing the message as a sequence of 0s and 1s. This can be accomplished by writing all numbers in binary, for example, or by using ASCII, as discussed in the previous section. But the message could also be a digitalized video or audio signal.

The key is a random sequence of 0s and 1s of the same length as the message. Once a key is used, it is discarded and never used again. The encryption consists of adding the key to the message mod 2, bit by bit. This process is often called **exclusive or**, and is denoted by XOR or $\oplus$. In other words,

we use the rules $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 1 = 0$. For example, if the message is 00101001 and the key is 10101100, we obtain the ciphertext as follows:

(plaintext) 00101001(key) ⊕10101100_(ciphertext) 10000101

Decryption uses the same key. Simply add the key onto the ciphertext: $10000101 \oplus 10101100 = 00101001$.

A variation is to leave the plaintext as a sequence of letters. The key is then a random sequence of shifts, each one between 0 and 25. Decryption uses the same key, but subtracts instead of adding the shifts.

This encryption method is completely unbreakable for a ciphertext-only attack. For example, suppose the ciphertext is

FIOWPSLQNTISJQL. The plaintext could be *wewillwinthewar* or it could be *theduckwantsout*. Each one is possible, along with all other messages of the same length. Therefore the ciphertext gives no information about the plaintext (except for its length). This will be made more precise in Section 4.4 and when we discuss Shannon's theory of entropy in Chapter 20.

If we have a piece of the plaintext, we can find the corresponding piece of the key, but it will tell us nothing about the remainder of the key. In most cases a chosen plaintext or chosen ciphertext attack is not possible. But such an attack would only reveal the part of the key used during the attack, which would not

be useful unless this part of the key were to be reused.

How do we implement this system, and where can it be used? The key can be generated in advance. Of course, there is the problem of generating a truly random sequence of 0s and 1s. One way would be to have some people sitting in a room flipping coins, but this would be too slow for most purposes. It is often suggested that we could take a Geiger counter and count how many clicks it makes in a small time period, recording a 0 if this number is even and 1 if it is odd, but care must be taken to avoid biases (see Exercise 12 in Chapter 5). There are other ways that are faster but not quite as random that can be used in

practice (see Chapter 5); but it is easy to see that quickly generating a good key is difficult. Once the key is generated, it can be sent by a trusted courier to the recipient. The message can then be sent when needed. It is reported that the "hot line" between Washington, D.C., and Moscow used one-time pads for secure communications between the leaders of the United States and the U.S.S.R. during the Cold War.

A disadvantage of the one-time pad is that it requires a very long key, which is expensive to produce and expensive to transmit. Once the key is used up, it is dangerous to reuse it for a second message; any knowledge of the first message gives knowledge of the second,

for example. Therefore, in most situations, various methods are used in which a small input can generate a reasonably random sequence of 0s and 1s, hence an "approximation" to a one-time pad. The amount of information carried by the courier is then several orders of magnitude smaller than the messages that will be sent. Two such methods, which are fast but not highly secure, are described in Chapter 5.

A variation of the one-time pad has been developed by Maurer, Rabin, Ding, and others. Suppose it is possible to have a satellite produce and broadcast several random sequences of bits at a rate fast enough that no computer can store more than a very small fraction of the outputs.

Alice wants to send a message to Bob. They use a public key method such as RSA (see Chapter 9) to agree on a method of sampling bits from the random bit streams. Alice and Bob then use these bits to generate a key for a one-time pad. By the time Eve has decrypted the public key transmission, the random bits collected by Alice and Bob have disappeared, so Eve cannot decrypt the message. In fact, since the encryption used a one-time pad, she can never decrypt it, so Alice and Bob have achieved everlasting security for their message. Note that bounded storage is an integral assumption for this procedure. The production and the accurate sampling of the bit streams are also important implementation issues.

4.3 Multiple Use of a One-Time Pad

Alice sends messages to Bob, Carla, and Dante. She encrypts each message with a one-time pad, but she's lazy and uses the same key for each message. In this section, we'll show how Eve can decrypt all three messages.

Suppose the messages are M1, M2, M3 and the key is K. The ciphertexts C1, C2, C3 are computed as $M_i \oplus K = C_i$. Eve computes

$$C1 \oplus C2 = (M1 \oplus K) \oplus (M2 \oplus K) = M1 \oplus M2.$$

Similarly, she obtains $M1 \oplus M3 = C1 \oplus C3$ and

$M2 \oplus M3 = C2 \oplus C3$. The key K has disappeared, and Eve's task is to deduce M1, M2, M3 from knowledge of $M1 \oplus M2$, $M1 \oplus M3$, $M2 \oplus M3$. The following example shows some basic ideas of the method.

Let's assume for simplicity that the messages are written in capital letters with spaces but with no other punctuation. The letters are converted to ASCII using

A=1000001, B=1000010, ⋯, Z=1011010, space =0100000

(the letters A to Z are the numbers 65 through 90 written in binary, and *space* is 32 in binary; see Table 4.1).

The XORs of the messages are

M1⊕M2=0000000 0001100 1100100 0001101 1100001 0011110 1111001 1100001 1100100 0011100 0001001 0001111 0010011 0010111 0000111M1⊕M3=0000001 1100101 1100001 0000101 1100001 0011011 1100101 00100111100101 1110011 0010001 0001100 0010110 0000100 0001101M2⊕M3=0000001 1101001 0000101 0001000 0000000 0000101 0011100 11100100000001 1101111 0011000 0000011 0000101 0010011 0001010.

Note that the first block of M1⊕M2 is `0000000`. This means that the first letter of M1 is the same as the first letter of M2. But the observation that makes the biggest difference for us is that "*space*" has a 0 as its leading bit, while all the letters have 1 as the leading bit. Therefore, if the leading bit of an XOR block is 1, then it arises from a letter XORed with "*space*."

For example, the third block of M1⊕M2 is `1100100` and the third block of M1⊕M3 is `1100001`. These can happen only from "space⊕1000100" and "space⊕1000001", respectively. It

follows easily that M1 has "*space*" as its third entry, M2 has 1000100=H as its third entry, and M3 has A as its third entry. Similarly, we obtain the 2nd, 7th, 8th, and 9th entries of each message.

The 5th entries cause a little more trouble: M1⊕M2 and M1⊕M3 tell us that there is "*space*" XORed with A, and M2⊕M3 tells us that the 5th entries of M2 and M3 are equal, but we could have "*space A A*" or "*A space space*." We need more information from surrounding letters to determine which it is.

To proceed, we use the fact that some letters are more common than others, and therefore certain XORs are more likely to arise from these than from

others. For example, the block 0010001 is more likely to arise from E⊕T=1000101⊕1010100 than from K⊕Z=1001011⊕1011010. The most common letters in English are E, T, A, O, I, N, S, H, R, D, L. Make a table of the XORs of each pair of these letters. If an XOR block occurs in this table, we guess that it comes from one of the pairs yielding this block. For example, `0001001` arises from A⊕H and from E⊕L, so we guess that the 11th block of M1⊕M2 comes from one of these two combinations. This might not be a correct guess, but we expect such guesses to be right often enough to yield a lot of information.

Rather than produce the whole table, we

give the blocks that we need and the combinations of these frequent letters that produce them:

0000001: D⊕E, H⊕I, R⊕S0000011: L⊕O0000100: A⊕E, H⊕L0000101: A

Let's look at the next-to-last block of each XOR. In M1⊕M2, we have `0010111`, which could come from D⊕S or E⊕R. In M1⊕M3, we have `0000100`, which could come from A⊕E or H⊕L. In M2⊕M3, we have `0010011`, which could come from A⊕R. The only combination consistent with these guesses is E, R, A for the next-to-last letters of M1, M2, M3, respectively. This type of reasoning, combined with the information obtained from the occurrence of spaces, yields the following progress (- indicates a space,

* indicates a letter to be determined):

M1=*E–**R–A–SE*RE*M2=*ID**LY–DOL*AR*M3=*–A**IERE–T*DA*

The first letter of M3 is a one-letter word. The XOR block `0000001` gives us the possibilities D, E, H, I, R, S, so we guess M3 starts with I. The XORs tell us that M1 and M2 start with H.

Now let's look at the 12th letters. The block `0001111` for M1⊕M2 suggests A⊕N. The block for M1⊕M3 suggests D⊕H and E⊕I. The block for M2⊕M3 suggests L⊕O. These do not have a common solution, so we know that one of the less common letters occurs in at least one of the messages. But M2 ends with DOL*AR*, and a good guess is that this should be DOLLARS. If so,

then the XOR information tells us that M1 ends in `SECRET`, and M3 ends in `TODAY`, both of which are words. So this looks like a good guess. Our progress so far is the following:

M1=HE–**R–A–SECRETM2=HID**LY–DOLLARSM3=I–A**IERE–TODAY

It is time to revisit the 5th letters. We already know that they are "*space A A*" or "*A space space*." The first possibility requires M1 to have two consecutive one-letter words, which seems unlikely. The second possibility means that M3 starts with the two words `I–A*`, so we can guess this is `I AM`. The XOR information tells us that M1 and M2 have H and E, respectively. We now have all the letters:

M1=HE–HAR–A–SECRETM2=HIDE–LY–DOLLARSM3=I–

AM–IERE–TODAY

Something seems to be wrong in the 6th column. These letters were deduced from the assumption that they all were common letters, and this must have been wrong. But at this point, we can make educated guesses. When we change I to H in M3, and make the corresponding changes in M1 and M2 required by the XORs, we get the final answer:

M1=HE–HAS–A–SECRETM2=HIDE–MY–DOLLARSM3=I–AM–HERE–TODAY.

These techniques can also be used when there are only two messages, but progress is usually slower. More possible combinations must be tried, and more false deductions need to be corrected during the decryption. The process can be automated. See

[Dawson-Nielsen].

The use of spaces and the restriction to capital letters made the decryption in the example easier. However, even if spaces are removed and more symbols are used, decryption is usually possible, though much more tedious.

During World War II, problems with production and distribution of one-time pads forced Russian embassies to reuse some keys. This was discovered, and part of the Venona Project by the U.S. Army's Signal Intelligence Service (the predecessor to NSA) was dedicated to deciphering these messages. Information obtained this way revealed several examples of Russian espionage, for example, in the Manhattan Project's

development of atomic bombs, in the State Department, and in the White House.

4.4 Perfect Secrecy of the One-Time Pad

Everyone knows that the one-time pad provides perfect secrecy. But what does this mean? In this section, we make this concept precise. Also, we know that it is very difficult in practice to produce a truly random key for a one-time pad. In the next section, we show quantitatively how biases in producing the key affect the security of the encryption.

In Section 20.4, we repeat some of the arguments of the present section and phrase them in terms of entropy and information theory.

The topics of this section and the next are part of the subject known as **Provable Security**. Rather than relying on intuition that a cryptosystem is secure, the goal is to isolate exactly what fundamental problems are the basis for its security. The result of the next section shows that the security of a one-time pad is based on the quality of the random number generator. In Section 10.5, we will show that the security of the ElGamal public key cryptosystem reduces to the difficulty of the Computational Diffie-Hellman problem, one of the fundamental problems related to discrete logarithms. In Section 12.3, we will use the Random Oracle Model to relate the security of a simple cryptosystem to the

noninvertibility of a one-way function. Since these fundamental problems have been well studied, it is easier to gauge the security levels of the cryptosystems.

First, we need to define **conditional probability**. Let's consider an example. We know that if it rains Saturday, then there is a reasonable chance that it will rain on Sunday. To make this more precise, we want to compute the probability that it rains on Sunday, given that it rains on Saturday. So we restrict our attention to only those situations where it rains on Saturday and count how often this happens over several years. Then we count how often it rains on both Saturday and Sunday. The ratio gives an estimate of the

desired probability. If we call A the event that it rains on Saturday and B the event that it rains on Sunday, then the **conditional probability of B given A** is

P(B|A)=P(A ∩ B)P(A), (4.1)

where P(A) denotes the probability of the event A. This formula can be used to define the conditional probability of one event given another for any two events A and B that have probabilities (we implicitly assume throughout this discussion that any probability that occurs in a denominator is nonzero).

Events A and B are **independent** if P(A ∩ B)=P(A)P(B). For example, if Alice flips a fair coin, let A be the event that the coin ends up Heads. If Bob rolls a fair six-sided die, let B be the event

that he rolls a 3. Then P(A)=1/2 and P(B)=1/6. Since all 12 combinations of {Head, Tail} and {1, 2, 3, 4, 5, 6} are equally likely, P(A ∩ B)=1/12, which equals P(A)P(B). Therefore, A and B are independent.

If A and B are independent, then

P(B|A)=P(A ∩ B)P(A)=P(A)P(B)P(A)=P(B),

which means that knowing that A happens does not change the probability that B happens. By reversing the steps in the above equation, we see that

A and B are independent⟺P(B|A)=P(B).

An example of events that are not independent is the original example, where A is the event that it rains on Saturday and B is the event that it rains

on Sunday, since $P(B|A)>P(B)$. (Unfortunately, a widely used high school algebra text published around 2005 gave exactly one example of independent events: A and B.)

How does this relate to cryptography? In a cryptosystem, there is a set of possible keys. Let's say we have N keys. If we have a perfect random number generator to choose the keys, then the probability that the key is k is $P(K=k)=1/N$. In this case we say that the key is chosen uniformly randomly. In any case, we assume that each key has a certain probability of being chosen. We also have various possible plaintexts m and each one has a certain probability $P(M=m)$. These probably do not all have

the same probability. For example, the message *attack at noon* is usually more probable than *two plus two equals seven*. Finally, each possible ciphertext c has a probability P(C=c).

We say that a cryptosystem has **perfect secrecy** if

$$P(M=m|C=c)=P(M=m)$$

for all possible plaintexts m and all possible ciphertexts c. In other words, knowledge of the ciphertext never changes the probability that a given plaintext occurs. This means that eavesdropping gives no advantage to Eve if she wants to guess the message.

We can now formalize what we claimed about the one-time pad.

Theorem

If the key is chosen uniformly randomly, then the one-time pad has perfect secrecy.

Proof. We need to show that $P(M=m|C=c)=P(M=m)$ for each pair m, c.

Let's say that there are N keys, each of which has probability 1/N. We start by showing that each possible ciphertext c also has probability 1/N. Start with any plaintext m. If c is the ciphertext, then the key is $k=m \oplus c$. Therefore, the probability that c is the ciphertext is the probability that $m \oplus c$ is the key, namely 1/N, since all keys have this probability. Therefore, we have proved that

$$P(C=c|M=m)=1/N$$

for each c and m.

We now combine the contributions from the various possibilities for m. Note that if we sum P(M=m) over all possible m, then

$$\sum mP(M=m)=1 \quad (4.2)$$

since this is the probability of the plaintext existing. Similarly, the event C=c can be split into the disjoint sets (C=c⋂M=m), which yields

$$P(C=c)=\sum mP(C=c \cap M=m)=\sum mP(C=c|M=m)P(M=m) \quad \text{(by (4.1))}=\sum m1NP(M=m)=1N \quad \text{(by (4.2))}.$$

Applying Equation (4.1) twice yields

$$P(M=m|C=c)P(C=c)=P(C=c\cap M=m)=P(C=c|M=m)P(M=m).$$

Since we have already proved that

$P(C=c)=1/N=P(C=c|M=m)$, we can multiply by N to obtain

$$P(M=m|C=c)=P(M=m),$$

which says that the one-time pad has perfect secrecy.

One of the difficulties with using the one-time pad is that the number of possible keys is as least as large the number of possible messages. Unfortunately, this is required for perfect secrecy:

Proposition

If a cryptosystem has perfect secrecy, then the number of possible keys is greater than or equal to the number of possible plaintexts.

Proof. Let M be the number of possible plaintexts and let N be the number of possible keys. Suppose M>N. Let c be a ciphertext. For each key k, decrypt c using the key k. This gives N possible plaintexts, and these are the only plaintexts that can encrypt to c. Since M>N, there is some plaintext m that is not a decryption of c. Therefore,

$$P(M=m|C=c)=0 \neq P(M=m).$$

This contradicts the assumption that the system has perfect secrecy. Therefore, $M \leq N$.

Example

Suppose a parent goes to the pet store to buy a pet for a child's birthday. The store sells 30 different pets with 3-letter

names (ant, bat, cat, dog, eel, elk, ...). The parent chooses a pet at random, encrypts its name with a shift cipher, and sends the ciphertext to let the other parent know what has been bought. The child intercepts the message, which is ZBL. The child hopes the present is a dog. Since DOG is not a shift of ZBL, the child realizes that the conditional probability

$$P(M=dog|C=ZBL)=0$$

and is disappointed. Since $P(M=dog)=1/30$ (because there are 30 equally likely possibilities), we have $P(M=dog|C=ZBL) \neq P(M=dog)$, so there is not perfect secrecy. This is because a given ciphertext has at most 26 possible corresponding plaintexts, so knowledge

of the ciphertext restricts the possibilities for the decryption. Then the child realizes that YAK is the only possible shift of ZBL, so $P(M=yak|C=ZBL)=1$. This does not equal $P(M=yak)$, so again we see that we don't have perfect secrecy. But now the child is happy, being the only one in the neighborhood who will have a yak as a pet.

4.5 Indistinguishability and Security

A basic requirement for a secure cryptosystem is **ciphertext indistinguishability**. This can be described by the following game:

CI Game: *Alice chooses two messages m0 and m1 and gives them to Bob. Bob randomly chooses b=0 or 1. He encrypts mb to get a ciphertext c, which he gives to Alice. Alice then guesses whether m0 or m1 was encrypted.*

By randomly guessing, Alice can guess correctly about 1/2 of the time. If there is no strategy where she guesses

correctly significantly more than 1/2 the time, then we say the cryptosystem has the ciphertext indistinguishability property.

For example, the shift cipher does not have this property. Suppose Alice chooses the two messages to be *CAT* and *DOG*. Bob randomly chooses one of them and sends back the ciphertext *PNG*. Alice observes that this cannot be a shift of *DOG* and thus concludes that Bob encrypted *CAT*.

When implemented in the most straightforward fashion, the RSA cryptosystem (see Chapter 9) also does not have the property. Since the encryption method is public, Alice can simply encrypt the two messages and

compare with what Bob sends her. However, if Bob pads the messages with random bits before encryption, using a good pseudorandom number generator, then Alice should not be able to guess correctly significantly more than 1/2 the time because she will not know the random bits used in the padding.

The one-time pad where the key is chosen randomly has ciphertext indistinguishability. Because Bob chooses b randomly,

$$P(m0)=P(m1)=12.$$

From the previous section, we know that

$$P(M=m0|C=c)=P(M=m0)=12$$

and

$P(M=m1|C=c)=P(M=m1)=12.$

Therefore, when Alice receives c, the two possibilities are equally likely, so the probability she guesses correctly is 1/2.

Because the one-time pad is too unwieldy for many applications, pseudorandom generators are often used to generate substitutes for one-time pads. In Chapter 5, we discuss some possibilities. For the present, we analyze how much such a choice can affect the security of the system.

A pseudorandom key generator produces N possible keys, with each possible key k having a probability $P(K=k)$. Usually, it takes an input, called a **seed**, and applies some

algorithm to produce a key that “looks random.” The seed is transmitted to the decrypter of the ciphertext, who uses the seed to produce the key and then decrypt. The seed is significantly shorter than the length of the key. While the key might have, for example, 1 million bits, the seed could have only 100 bits, which makes transmission much more efficient, but this also means that there are fewer keys than with the one-time pad. Therefore, Proposition 4.4 says that perfect secrecy is not possible.

If the seed had only 20 bits, it would be possible to use all of the seeds to generate all possible keys. Then, given a ciphertext and a plaintext, it would be easy to see if there is a key that encrypts

the plaintext to the ciphertext. But with a seed of 100 bits, it is infeasible to list all 2100 seeds and find the corresponding keys. Moreover, with a good pseudorandom key generator, it should be difficult to see whether a given key is one that could be produced from some seed.

To evaluate a pseudorandom key generator, Alice (the adversary) and Bob play the following game:

R Game: *Bob flips a fair coin. If it's Heads, he chooses a number r uniformly randomly from the keyspace. If it's Tails, he chooses a pseudorandom key r. Bob sends r to Alice. Alice guesses whether r was chosen randomly or pseudorandomly.*

Of course, Alice could always guess that it's random, for example, or she could flip her own coin and use that for her guess. In these cases, her probability of guessing correctly is 1/2. But suppose she knows something about the pseudorandom generator (maybe she has analyzed its inner workings, for example). Then she might be able to recognize sometimes that r looks like something the pseudorandom generator could produce (of course, the random generator could also produce it, but with lower probability since it has many more possible outputs). This could occasionally give Alice a slight edge in guessing. So Alice's overall probability of winning could increase slightly.

In an extreme case, suppose Alice knows that the pseudorandom number generator always has a 1 at the beginning of its output. The true random number generator will produce such an output with probability 1/2. If Alice sees this initial 1, she guesses that the output is from the pseudorandom generator. And if this 1 is not present, Alice knows that r is random. Therefore, Alice guesses correctly with probability 3/4. (This is Exercise 9.)

We write

$$P(\text{Alice is correct}) = 12 + \epsilon.$$

A good pseudorandom generator should have ϵ very small, no matter what strategy Alice uses.

But will a good pseudorandom key generator work in a one-time pad? Let's test it with the CI Game. Suppose Charles is using a pseudorandom key generator for his one-time pad and he is going to play the CI game with Alice. Moreover, suppose Alice has a strategy for winning this CI Game with probability $12+\epsilon$ (if this happens, then Charles's implementation is not very good). We'll show that, under these assumptions, Alice can play the R game with Bob and win with probability $12+\epsilon 2$.

For example, suppose that the pseudorandom number generator is such that Alice has probability at most 0.51 of winning the R Game. Then we must

have $\epsilon/2 \leq .01$, so $\epsilon \leq .02$. This means that the probability that Charles wins the CI game is at most 0.52. In this way, we conclude that if the random number generator is good, then its use in a one-time pad is good.

Here is Alice's strategy for the R game. Alice and Bob do the following:

1. Bob flips a fair coin, as in the R game, and gives r to Alice. Alice wants to guess whether r is random or pseudorandom.

2. Alice uses r to play the CI game with Charles.

    1. She calls up Charles, who chooses messages m0 and m1 and gives them to Alice.

    2. Alice chooses b=0 or 1 randomly.

    3. She encrypts mb using the key r to obtain the ciphertext $c=mb \oplus r$.

4. She sends c to Charles.

5. Charles makes his guess for b and succeeds with probability $12+\epsilon$.

3. Alice now uses Charles's guess to finish playing the R Game.

4. If Charles guessed b correctly, her guess to Bob is that r was pseudorandom.

5. If Charles guessed incorrectly, her guess to Bob is that r was random.

There are two ways that Alice wins the R Game. One is when Charles is correct and r is pseudorandom, and the other is when Charles is incorrect and r is random.

The probability that Charles is correct when r is pseudorandom is $12+\epsilon$, by assumption. This means that

$$P(\text{Charles is correct} \cap \text{r is pseudorandom}) = P(\text{Charles is correct} | \text{r is}$$

pseudorandom)P(r is pseudorandom)=(12+ϵ)(1/2).

If r is random, then Alice encrypted mb with a true one-time pad, so Charles succeeds half the time and fails half the time. Therefore,

P(Charles is incorrect∩r is random)=P(Charles is incorrect|r is random)P(r is random)=(12)(1/2).

Putting the previous two calculations together, we see that the probability that Alice wins the R Game is

(12+ϵ)(1/2)+(12)(1/2)=12+12ϵ,

as we claimed.

The preceding shows that if we design a good random key generator, then an adversary can gain only a very slight advantage in using a ciphertext to distinguish between two plaintexts.

Unfortunately, there are not good ways to prove that a given pseudorandom number generator is good (this would require solving some major problems in complexity theory), but knowledge of where the security of the system lies is significant progress.

For a good introduction to cryptography via the language of computational security and proofs of security, see [Katz and Lindell].

4.6 Exercises

1. Alice is learning about the shift cipher. She chooses a random three-letter word (so all three-letter words in the dictionary have the same probability) and encrypts it using a shift cipher with a randomly chosen key (that is, each possible shift has probability 1/26). Eve intercepts the ciphertext *mxp*.

    1. Compute P(M=cat|C=mxp). (Hint: Can *mxp* shift to *cat*?)

    2. Use your result from part (a) to show that the shift cipher does not have perfect secrecy (this is also true because there are fewer keys than ciphertexts; see the proposition at the end of the first section).

2. Alice is learning more advanced techniques for the shift cipher. She now chooses a random five-letter word (so all five-letter words in the dictionary have the same probability) and encrypts it using a shift cipher with a randomly chosen key (that is, each

possible shift has probability 1/26). Eve intercepts the ciphertext *evire*. Show that

P(M=arena|C=evire)=1/2.

(Hint: Look at Exercise 1 in Chapter 2.)

3. Suppose a message m is chosen randomly from the set of all five-letter English words and is encrypted using an affine cipher mod 26, where the key is chosen randomly from the 312 possible keys. The ciphertext is HHGZC. Compute the conditional probability Prob(m=HELLO|c=HHGZC). Use the result of this computation to determine whether of not affine ciphers have perfect secrecy.

4. Alice is learning about the Vigenère cipher. She chooses a random six-letter word (so all six-letter words in the dictionary have the same probability) and encrypts it using a Vigenère cipher with a randomly chosen key of length 3 (that is, each possible key has probability 1/263). Eve intercepts the ciphertext *eblkfg*.

    1. Compute the conditional probability P(M=attack|C=eblkfg).

    2. Use your result from part (a) to show that the Vigenère cipher does not have perfect secrecy.

5. Alice and Bob play the following game (this is the CI Game of Section 4.5). Alice chooses two two-letter words m0 and m1 and gives them to Bob. Bob randomly chooses b=0 or 1. He encrypts mb using a shift cipher (with a randomly chosen shift) to get a ciphertext c, which he gives to Alice. Alice then guesses whether m0 or m1 was encrypted.

    1. Alice chooses m0=HI and m1=NO. What is the probability that Alice guesses correctly?

    2. Give a choice of m0 and m1 that Alice can make so that she is guaranteed to be able to guess correctly.

6. Bob has a weak pseudorandom generator that produces N different M-bit keys, each with probability 1/N. Alice and Bob play Game R. Alice makes a list of the N possible pseudorandom keys. If the number r that Bob gives to her is on this list, she guesses that the number is chosen pseudorandomly. If it is not on the list, she guess that it is random.

    1. Show that

       P(r is on the list|r is random)=N2M

       and

P(r is on the list|r is pseudorandom)=1.

2. Show that

   P(r is random⋂r is on the list)=12N2MP(r is random⋂r is not on the list)=121−N2MP(r is pseudorandom⋂r is on the list)=12P(r is pseudorandom⋂r is not on the list)=0.

3. Show that Alice wins with probability

   121−N2M+12.

4. Show that if N/2M=1−ϵ then Alice wins with probability 12+12ϵ. (This shows that if the pseudorandom generator misses a fraction of the possible keys, then Alice has an advantage in the game, provided that she can make a list of all possible outputs of the generator. Therefore, it is necessary to make N large enough that making such a list is infeasible.)

7. Suppose Alice knows that Bob's pseudorandom key generator has a slight bias and that with probability 51% it produces a key with more 1's than 0's. Alice and Bob play the CI Game. Alice chooses messages m0=000⋯0 and m1=111⋯1 to

Bob, who randomly chooses $b \in \{0, 1\}$ and encrypts m_b with a one-time pad using his pseudorandom key generator. He gives the ciphertext $c = m_b \oplus r$ (where r is his pseudorandom key) to Alice. Alice computes $s = c \oplus m_0$. If s has more 1's than 0's, she guesses that $b=0$. If not, she guesses that $b=1$. For simplicity in the following, we assume that the message lengths are odd (so there cannot be the same number of 1's and 0's).

1. Show that exactly one of $m_0 \oplus r$ and $m_1 \oplus r$ has more 1's than 0's.

2. Show that $P(s \text{ has more 1's} | b=0) = .51$

3. Show that $P(s \text{ has fewer 1's} | b=1) = .51$

4. Show that Alice has a probability .51 of winning.

8. In the one-time pad, suppose that some plaintexts are more likely than others. Show that the key and the ciphertext are not independent. That is, show that there is a key k and a ciphertext c such that

   $$P(C=c \cap K=k) \neq P(C=c)P(K=k).$$

   (Hint: The right-hand side of this equation is independent of k and c. What about the left-hand side?)

9. Alice and Bob are playing the R Game. Suppose Alice knows that Bob's pseudorandom number generator always has a 1 at the beginning of its output. If Alice sees this initial 1, she guesses that the output is from the pseudorandom generator. And if this 1 is not present, Alice knows that r is random. Show that Alice guesses correctly with probability 3/4.

10. Suppose Bob uses a pseudorandom number generator to produce a one-time pad, but the generator has a slight bias, so each bit it produces has probability 51% of being 1 and only 49% of being 0. What strategy can Alice use so that she expects to win the CI Game more than half the time?

11. At the end of the semester, the professor randomly chooses and sends one of two possible messages:

    m0=YOUPASSEDandm1=YOUFAILED.

    To add to the excitement, the professor encrypts the message using one of the following methods:

    1. Shift cipher

    2. Vigenère cipher with key length 3

    3. One-time pad.

You receive the ciphertext and want to decide whether the professor sent m0 or m1. For each method (a), (b), (c), explain how to decide which message was sent or explain why it is impossible to decide. You may assume that you know which method is being used. (For the Vigenère, do not do frequency analysis; the message is too short.)

12. On Groundhog Day, the groundhog randomly chooses and sends one of two possible messages:

    m0=SIXMOREWEEKSOFWINTERm1=SPRINGARRIVESINMA

    To add to the mystery, the groundhog encrypts the message using one of the following methods: shift cipher, Vigenère cipher with key length 4 (using four distinct shifts), one-time pad. For each of the following ciphertexts, determine which encryption methods could have produced that ciphertext, and for each of these possible encryption methods, decrypt the message or explain why it is impossible to decrypt.

    1. *ABCDEFGHIJKLMNOPQRST*

    2. *UKZOQTGYGGMUQHYKPVGT*

    3. *UQUMPHDVTJYIUJQQCSFL.*

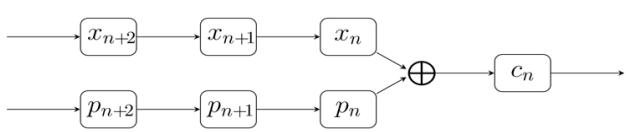
13. Alice encrypts the messages M1 and M2 with the

same one-time pad using only capital letters and spaces, as in Section 4.3. Eve knows this, intercepts the ciphertexts C1 and C2, and also learns that the decryption of M1 is *THE LETTER * ON THE MAP GIVES THE LOCATION OF THE TREASURE*. Unfortunately for Eve, she cannot read the missing letter *. However, the 12th group of seven bits in C1 is 1001101 and the 12th group in C2 is 0110101. Find the missing letter.

Chapter 5 Stream Ciphers

The one-time pad provides a strong form of secrecy, but since key transmission is difficult, it is desirable to devise substitutes that are easier to use. Stream ciphers are one way of achieving this goal. As in the one-time pad, the plaintext is written as a string of bits. Then a binary keystream is generated and XORed with the plaintext to produce the ciphertext.

Figure 5.1 Stream cipher encryption



pn= plaintext bit, xn= key bit, cn= ciphertext bit.

Figure 5.1 Full Alternative Text

For the system to be secure, the keystream needs to approximate a random sequence, so we need a good source of random-looking bits. In Section 5.1, we discuss pseudorandom number generators. In Sections 5.2 and 5.3, we describe two commonly used stream ciphers and the pseudorandom number generators that they use. Although they have security weaknesses, they give an idea of methods that can be used.

In the next chapter, we discuss block ciphers and various modes of operations. Some of the most secure

stream ciphers are actually good block ciphers used, for example, in OFB or CTR mode. See Subsections 6.3.4 and 6.3.5.

There is one problem that is common to all stream ciphers that are obtained by XORing pseudorandom numbers with plaintext and is one of the reasons that authentication and message integrity checks are added to protect communications. Suppose Eve knows where the word “good” occurs in a plaintext that has been encrypted with a stream cipher. If she intercepts the ciphertext, she can XOR the bits for good $\oplus$ evil at the appropriate place in the ciphertext before continuing the transmission of the ciphertext. When the

ciphertext is decrypted, “good” will be changed to “evil.” This type of attack was one of the weaknesses of the WEP system, which is discussed in Section 14.3.

5.1 Pseudorandom Bit Generation

The one-time pad and many other cryptographic applications require sequences of random bits. Before we can use a cryptographic algorithm, such as DES (Chapter 7) or AES (Chapter 8), it is necessary to generate a sequence of random bits to use as the key.

One way to generate random bits is to use natural randomness that occurs in nature. For example, the thermal noise from a semiconductor resistor is known to be a good source of randomness. However, just as flipping coins to produce random bits would not be

practical for cryptographic applications, most natural conditions are not practical due to the inherent slowness in sampling the process and the difficulty of ensuring that an adversary does not observe the process. We would therefore like a method for generating randomness that can be done in software. Most computers have a method for generating random numbers that is readily available to the user. For example, the standard C library contains a function *rand()* that generates pseudorandom numbers between 0 and 65535. This pseudorandom function takes a **seed** as input and produces an output bitstream.

The *rand()* function and many other pseudorandom number generators are

based on linear congruential generators. A **linear congruential generator** produces a sequence of numbers x1, x2, ⋯, where

$$x_n = a x_{n-1} + b \pmod{m}.$$

The number x0 is the initial seed, while the numbers a, b, and m are parameters that govern the relationship. The use of pseudorandom number generators based on linear congruential generators is suitable for experimental purposes, but is highly discouraged for cryptographic purposes. This is because they are predictable (even if the parameters a, b, and m are not known), in the sense that an eavesdropper can use knowledge of some bits to predict future bits with fairly high probability. In fact, it has

been shown that any polynomial congruential generator is cryptographically insecure.

In cryptographic applications, we need a source of bits that is nonpredictable. We now discuss two ways to create such nonpredictable bits.

The first method uses one-way functions. These are functions f(x) that are easy to compute but for which, given y, it is computationally infeasible to solve y=f(x) for x. Suppose that we have such a one-way function f and a random seed s. Define xj=f(s+j) for j=1, 2, 3, …. If we let bj be the least significant bit of xj, then the sequence b0, b1, ⋯ will often be a pseudorandom sequence of bits (but see Exercise 14). This method

of random bit generation is often used, and has proven to be very practical. Two popular choices for the one-way function are DES (Chapter 7) and SHA, the Secure Hash Algorithm (Chapter 11). As an example, the cryptographic pseudorandom number generator in the OpenSSL toolkit (used for secure communications over the Internet) is based on SHA.

Another method for generating random bits is to use an intractable problem from number theory. One of the most popular cryptographically secure pseudorandom number generators is the **Blum-Blum-Shub (BBS) pseudorandom bit generator**, also known as the quadratic residue

generator. In this scheme, one first generates two large primes p and q that are both congruent to 3 mod 4. We set n=pq and choose a random integer x that is relatively prime to n. To initialize the BBS generator, set the initial seed to $x_0 \equiv x^2 \pmod{n}$. The BBS generator produces a sequence of random bits $b_1, b_2, \cdots$ by

1. $x_j \equiv x_{j-1}^2 \pmod{n}$
2. b_j is the least significant bit of x_j.

Example

Let

$p=24672462467892469787$ and $q=396736894567834589803$, $n=978847614085311079416885521741371578196 1$.

Take $x=873245647888478349013$. The initial seed is

x0 ≡ x2(mod n) ≡ 884529871047878009708991774601012286317 2.

The values for x1, x2, ⋯x8 are

x1 ≡ 7118894281131329522745962455498123822408x2 ≡ 314517460888889

Taking the least significant bit of each of these, which is easily done by checking whether the number is odd or even, produces the sequence b1, ⋯, b8=0, 1, 1, 1, 0, 0, 0, 0.

The Blum-Blum-Shub generator is very likely unpredictable. See [Blum-Blum-Shub]. A problem with BBS is that it can be slow to calculate. One way to improve its speed is to extract the k least significant bits of xj. As long as k ≤ log2 log2n, , this seems to be cryptographically secure.

5.2 Linear Feedback Shift Register Sequences

Note: **In this section, all congruences are mod 2**.

In many situations involving encryption, there is a trade-off between speed and security. If one wants a very high level of security, speed is often sacrificed, and vice versa. For example, in cable television, many bits of data are being transmitted, so speed of encryption is important. On the other hand, security is not usually as important since there is rarely an economic advantage to mounting an expensive attack on the system.

In this section, we describe a method that could be used when speed is more important than security. However, the real use is as one building block in more complex systems.

The sequence

0100001001011001111100011011101010000100101100111 1

can be described by giving the initial values

$x1 \equiv 0,\ x2 \equiv 1,\ x3 \equiv 0,\ x4 \equiv 0,\ x5 \equiv 0$

and the linear recurrence relation

$xn+5 \equiv xn+xn+2 (\mathrm{mod}\ 2).$

This sequence repeats after 31 terms.

For another example, see Example 18 in the Computer Appendices.

More generally, consider a linear recurrence relation of **length** m:

$$x_{n+m} \equiv c_0 x_n + c_1 x_{n+1} + \cdots + c_{m-1} x_{n+m-1} \pmod 2,$$

where the coefficients c0, c1, … are integers. If we specify the **initial values**

$$x_1, x_2, \ldots, x_m,$$

then all subsequent values of xn can be computed using the recurrence. The resulting sequence of 0s and 1s can be used as the key for encryption. Namely, write the plaintext as a sequence of 0s and 1s, then add an appropriate number of bits of the key sequence to the plaintext mod 2, bit by bit. For example, if the plaintext is 1011001110001111 and the key sequence is the example given previously, we have

(plaintext)1011001110001111(key)
⊕0100001001011001_(ciphertext)1111000111010110

Decryption is accomplished by adding the key sequence to the ciphertext in exactly the same way.

One advantage of this method is that a key with large period can be generated using very little information. The long period gives an improvement over the Vigenère method, where a short period allowed us to find the key. In the above example, specifying the initial vector {0, 1, 0, 0, 0} and the coefficients {1, 0, 1, 0, 0} yielded a sequence of period 31, so 10 bits were used to produce 31 bits. It can be shown that the recurrence

$x_{n+31} \equiv x_n + x_{n+3}$

and any nonzero initial vector will produce a sequence with period 231−1=2147483647. Therefore, 62 bits produce more than two billion bits of key. This is a great advantage over a one-time pad, where the full two billion bits must be sent in advance.

This method can be implemented very easily in hardware using what is known as a **linear feedback shift register** (LFSR) and is very fast. In Figure 5.2 we depict an example of a linear feedback shift register in a simple case. More complicated recurrences are implemented using more registers and more XORs.

Figure 5.2 A Linear

Feedback Shift Register Satisfying xn+3=xn+1+xn.

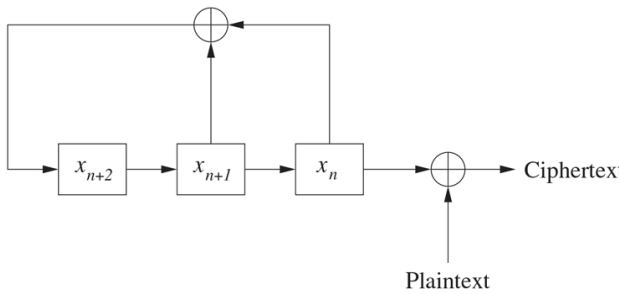


Figure 5.2 Full Alternative Text

For each increment of a counter, the bit in each box is shifted to other boxes as indicated, with ⊕ denoting the addition mod 2 of the incoming bits. The output, which is the bit xn, is added to the next bit of plaintext to produce the ciphertext. The diagram in Figure 5.2 represents the recurrence xn+3 ≡ xn +1+xn. Once the initial values x1, x2, x3 are specified, the machine produces the subsequent bits very efficiently.

Unfortunately, the preceding encryption method succumbs easily to a known plaintext attack. More precisely, if we know only a few consecutive bits of plaintext, along with the corresponding bits of ciphertext, we can determine the recurrence relation and therefore compute all subsequent bits of the key. By subtracting (or adding; it's all the same mod 2) the plaintext from the ciphertext mod 2, we obtain the bits of the key. Therefore, for the rest of this discussion, we will ignore the ciphertext and plaintext and assume we have discovered a portion of the key sequence. Our goal is to use this portion of the key to deduce the coefficients of the recurrence and consequently compute the rest of the key.

For example, suppose we know the initial segment 011010111100 of the sequence 011010111100010011010111…, which has period 15, and suppose we know it is generated by a linear recurrence. How do we determine the coefficients of the recurrence? We do not necessarily know even the length, so we start with length 2 (length 1 would produce a constant sequence). Suppose the recurrence is xn+2=c0xn+c1xn+1. Let n=1 and n=2 and use the known values x1=0, x2=1, x3=1, x4=0. We obtain the equations

1 ≡ c0·0+c1·1(n=1).0 ≡ c0·1+c1·1(n=2).

In matrix form, this is

(0111) (c0c1) ≡ (10).

The solution is c0=1, c1=1, so we guess that the recurrence is xn+2 ≡ xn+xn+1. Unfortunately, this is not correct since x6 ≡ x4+x5. Therefore, we try length 3. The resulting matrix equation is

(011110101) (c0c1c2) ≡ (010).

The determinant of the matrix is 0 mod 2; in fact, the equation has no solution. We can see this because every column in the matrix sums to 0 mod 2, while the vector on the right does not.

Now consider length 4. The matrix equation is

(0110110110100101) (c0c1c2c3) ≡ (1011).

The solution is c0=1, c1=1, c2=0, c3=0. The resulting recurrence is now conjectured to be

xn+4 ≡ xn+xn+1.

A quick calculation shows that this generates the remaining bits of the piece of key that we already know, so it is our best guess for the recurrence that generates the key sequence.

What happens if we try length 5? The matrix equation is

01101110101010101011101110c0c1c2c3c4 ≡ 01111.

The determinant of the matrix is 0 mod 2. Why? Notice that the last row is the sum of the first and second rows. This is a consequence of the recurrence relation: x5 ≡ x1+x2, x6 ≡ x2+x3, etc. As in linear algebra with real or complex numbers, if one row of a matrix is a linear combination of other rows, then the determinant is 0.

Similarly, if we look at the 6×6 matrix, we see that the 5th row is the sum of the first and second rows, and the 6th row is the sum of the second and third rows, so the determinant is 0 mod 2. In general, when the size of the matrix is larger than the length of the recurrence relation, the relation forces one row to be a linear combination of other rows, hence the determinant is 0 mod 2.

The general situation is as follows. To test for a recurrence of length m, we assume we know x1, x2, …, x2m. The matrix equation is

x1x2⋯xmx2x3⋯xm+1⋮⋮⋱⋮xmxm+1⋯x2m−1c0c1⋮cm−1 ≡ xm+1xm+2⋮x2m.

We show later that the matrix is invertible mod 2 if and only if there is

no linear recurrence of length less than m that is satisfied by x1, x2, …, x2m−1.

A strategy for finding the coefficients of the recurrence is now clear. Suppose we know the first 100 bits of the key. For m=2, 3, 4, …, form the m×m matrix as before and compute its determinant. If several consecutive values of m yield 0 determinants, stop. The last m to yield a nonzero (i.e., 1 mod 2) determinant is probably the length of the recurrence. Solve the matrix equation to get the coefficients c0, …, cm−1. It can then be checked whether the sequence that this recurrence generates matches the sequence of known bits of the key. If not, try larger values of m.

Suppose we don’t know the first 100

bits, but rather some other 100 consecutive bits of the key. The same procedure applies, using these bits as the starting point. In fact, once we find the recurrence, we can also work backwards to find the bits preceding the starting point.

Here is an example. Suppose we have the following sequence of 100 bits:

1001100100111000110001010001111011001111101010100101101101011000

The first 20 determinants, starting with m=1, are

1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.

A reasonable guess is that m=8 gives the last nonzero determinant. When we solve the matrix equation for the coefficients we get

{c0, c1, …, c7}={1, 1, 0, 0, 1, 0, 0, 0},

so we guess that the recurrence is

xn+8 ≡ xn+xn+1+xn+4.

This recurrence generates all 100 terms of the original sequence, so we have the correct answer, at least based on the knowledge that we have.

Suppose that the 100 bits were in the middle of some sequence, and we want to know the preceding bits. For example, suppose the sequence starts with x17, so x17=1, x18=0, x19=0, …. Write the recurrence as

xn ≡ xn+1+xn+4+xn+8

(it might appear that we made some sign errors, but recall that we are working mod 2, so −xn ≡ xn and −xn+8 ≡ xn+8).

Letting n=16 yields

$$x16 \equiv x17+x20+x24 \equiv 1+0+1 \equiv 0.$$

Continuing in this way, we successively determine x15, x14, …, x1.

For more examples, see Examples 19 and 20 in the Computer Appendices.

We now prove the result we promised.

Proposition

Let x1, x2, x3, … be a sequence of bits produced by a linear recurrence mod 2. For each n ≥ 1, let

$$Mn=x1x2\cdots xnx2x3\cdots xn+1\vdots\ddots\vdots xnxn+1\cdots x2n-1.$$

Let N be the length of the shortest recurrence that generates the sequence

x1, x2, x3, …. Then
$\det(MN) \equiv 1 \pmod 2$ and
$\det(Mn) \equiv 0 \pmod 2$ for all $n>N$.

Proof. We first make a few remarks on the length of recurrences. A sequence could satisfy a length 3 relation such as $x_{n+3} \equiv x_{n+2}$. It would clearly then also satisfy shorter relations such as $x_{n+1}=x_n$ (at least for $n \geq 2$). However, there are less obvious ways that a sequence could satisfy a recurrence of length less than expected. For example, consider the relation $x_{n+4} \equiv x_{n+3}+x_{n+1}+x_n$. Suppose the initial values of the sequence are 1, 1, 0, 1. The recurrence allows us to compute subsequent terms: 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1…. It is easy to see that the sequence

satisfies $x_{n+2} \equiv x_{n+1}+x_n$.

If there is a recurrence of length N and if n>N, then one row of the matrix Mn is congruent mod 2 to a linear combination of other rows. For example, if the recurrence is $x_{n+3}=x_{n+2}+x_n$, then the fourth row is the sum of the first and third rows. Therefore, $\det(M_n) \equiv 0 \pmod 2$ for all n>N.

Now suppose $\det(MN) \equiv 0 \pmod 2$. Then there is a nonzero row vector $\bar{b}=(b_0, \ldots, b_{N-1})$ such that $\bar{b}MN \equiv 0$. We'll show that this gives a recurrence relation for the sequence $x_1, x_2, x_3, \ldots$ and that the length of this relation is less than N. This contradicts the assumption that N is smallest. This contradiction implies that $\det(MN) \equiv 1 \pmod 2$.

Let the recurrence of length N be

$$x_{n+N} \equiv c_0 x_n + \cdots + c_{N-1} x_{n+N-1}.$$

For each $i \geq 0$, let

$$M(i) = \begin{pmatrix} x_{i+1} & x_{i+2} & \cdots & x_{i+N} \\ x_{i+2} & x_{i+3} & \cdots & x_{i+N+1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{i+N} & x_{i+N+1} & \cdots & x_{i+2N-1} \end{pmatrix}.$$

Then $M(0) = M^N$. The recurrence relation implies that

$$M(i) \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{pmatrix} \equiv \begin{pmatrix} x_{i+N+1} \\ x_{i+N+2} \\ \vdots \\ x_{i+2N} \end{pmatrix},$$

which is the last column of $M(i+1)$.

By the choice of $\bar{b}$, we have $\bar{b}M(0) = \bar{b}M^N = 0$. Suppose that we know that $\bar{b}M(i) = 0$ for some i. Then

$$\bar{b} \begin{pmatrix} x_{i+N+1} \\ x_{i+N+2} \\ \vdots \\ x_{i+2N} \end{pmatrix} \equiv \bar{b}M(i) \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{pmatrix} \equiv 0.$$

Therefore, $\bar{b}$ annihilates the last column of $M(i+1)$. Since the remaining columns of $M(i+1)$ are columns of $M(i)$,

we find that $b^{-}M(i+1) \equiv 0$. By induction, we obtain $b^{-}M(i) \equiv 0$ for all $i \geq 0$.

Let $n \geq 1$. The first column of $M(n-1)$ yields

$$b_0x_{n}+b_1x_{n+1}+\cdots+b_{N-1}x_{n+N-1} \equiv 0.$$

Since b^{-} is not the zero vector, $b_j \neq 0$ for at least one j. Let m be the largest j such that $b_j \neq 0$, which means that $b_m=1$. We are working mod 2, so $b_mx_{n+m} \equiv -x_{n+m}$. Therefore, we can rearrange the relation to obtain

$$x_{n+m} \equiv b_0x_{n}+b_1x_{n+1}+\cdots+b_{m-1}x_{n+m-1}.$$

This is a recurrence of length m. Since $m \leq N-1$, and N is assumed to be the shortest possible length, we have a contradiction. Therefore, the assumption that $\det(MN) \equiv 0$ must be false, so

$\det(MN) \equiv 1$. This completes the proof.

Finally, we make a few comments about the period of a sequence. Suppose the length of the recurrence is m. Any m consecutive terms of the sequence determine all future elements, and, by reversing the recurrence, all previous values, too. Clearly, if we have m consecutive 0s, then all future values are 0. Also, all previous values are 0. Therefore, we exclude this case from consideration. There are $2^m - 1$ strings of 0s and 1s of length m in which at least one term is nonzero. Therefore, as soon as there are more than $2^m - 1$ terms, some string of length m must occur twice, so the sequence repeats. The period of the sequence is at most $2^m - 1$.

Associated to a recurrence $x_{n+m} \equiv c_0 x_n + c_1 x_{n+1} + \cdots + c_{m-1} x_{n+m-1} \pmod 2$, there is a polynomial

$$f(T) = T^m - c_{m-1} T^{m-1} - \cdots - c_0.$$

If $f(T)$ is irreducible mod 2 (this means that it is not congruent to the product of two lower-degree polynomials), then it can be shown that the period divides $2^m - 1$. An interesting case is when $2^m - 1$ is prime (these are called Mersenne primes). If the period isn't 1, that is, if the sequence is not constant, then the period in this special case must be maximal, namely $2^m - 1$ (see Section 3.11). The example where the period is $2^{31} - 1$ is of this type.

Linear feedback shift register sequences have been studied extensively. For

example, see [Golomb] or [van der Lubbe].

One way of thwarting the above attack is to use nonlinear recurrences, for example,

$$x_{n+3} \equiv x_{n+2}x_n + x_{n+1}.$$

Moreover, a look-up table that takes inputs x_n, x_{n+1}, x_{n+2} and outputs a bit x_{n+3} could be used, or several LFSRs could be combined nonlinearly and some of these LFSRs could have irregular clocking. Generally, these systems are somewhat harder to break. However, we shall not discuss them here.

5.3 RC4

RC4 is a stream cipher that was developed by Rivest and has been widely used because of its speed and simplicity. The algorithm was originally secret, but it was leaked to the Internet in 1994 and has since been extensively analyzed. In particular, certain statistical biases were found in the keystream it generated, especially in the initial bits. Therefore, often a version called RC4-drop[n] is used, in which the first n bits are dropped before starting the keystream. However, this version is still not recommended for situations requiring high security.

To start the generation of the keystream for RC4, the user chooses a key, which is a binary string between 40 and 256 bits long. This is put into the Key Scheduling Algorithm. It starts with an array S consisting of the numbers from 0 to 255, regarded as 8-bit bytes, and outputs a permutation of these entries, as follows:

Algorithm 1 RC4 Key Scheduling Algorithm

- 1: **for** i from 0 to 255 **do**
- 2: S[i]:=i
- 3: j:=0
- 4: **for** i from 0 to 255 **do**
- 5: j:=(j+S[i]+key[i mod keylength]) mod 256
- 6: swap(S[i], S[j])

This algorithm starts by initializing the entries in S as S[i]=i for i running from 0 through 255. Suppose the user-supplied key is 10100000⋯ (let's say the key length is 40).

The algorithm starts with j=0 and i=0. The value of j is updated to j +S[i]+key[i mod 40]=0+0+1=1. Then S[i]=S[0]=0 and S[j]=S[1]=1 are swapped, so now S[0]=1 and S[1]=0.

We now move to i=1. The value j=1 is updated to 1+0+key[1]=1, so S[1] is swapped with itself, which means it is not changed here.

We now move to i=2. The value j=1 is updated to 1+2+key[2]=4, so S[2]=2 is swapped with S[4]=4, yielding S[2]=4

and S[4]=2.

We now move to i=3. The value j=4 is updated to 4+3+0=7, so S[3] and S[7] are swapped, yielding S[3]=7 and S[7]=3.

Let's look at one more value of i, namely, i=4. The value j=7 is updated to 7+2+0=9 (recall that S[4] became 2 earlier), and we obtain S[4]=9 and S[9]=2.

This process continues through i=255 and yields an array S of length 256 consisting of a permutation of the numbers from 0 through 255.

The array S is entered into the Pseudorandom Generation Algorithm.

Algorithm 2 RC4 Pseudorandom Generation Algorithm (PRGA)

- i:=0
- j:=0
- **while** GeneratingOutput: **do**
    - i:=(i+1) mod 256
    - j:=(j+S[i]) mod 256
    - swap(S[i], S[j])
    - output S[(S[i]+S[j]) mod 256]

This algorithm runs as long as needed and each round outputs a number between 0 and 255, regarded as an 8-bit byte. This byte is XORed with the corresponding byte of the plaintext to yield the ciphertext.

Weaknesses. Generally, the keystream that is output by a stream cipher should be difficult to distinguish from a randomly generated bitstream. For example, the R Game (see Section 4.5) could be played, and the probability of winning should be negligibly larger than 1/2. For RC4, there are certain observable biases. The second byte in the output should be 0 with probability 1/256. However, Mantin and Shamir [Mantin-Shamir] showed that this byte is 0 with twice that probability. Moreover, they found that the probability that the first two bytes are simultaneously 0 is 3/2562 instead of the expected 1/2562.

Biases have also been found in the state

S that is output by the Key Scheduling Algorithm. For example, the probability that S[0]=1 is about 37% larger than the expected probability of 1/256, while the probability that S[0]=255 is 26% less than expected.

Although any key length from 40 to 255 bits can be chosen, the use of small key sizes is not recommended because the algorithm can succumb to a brute force attack.

5.4 Exercises

1. A sequence generated by a length 3 recurrence starts 001110. Find the next four elements of the sequence.

2. The LFSR sequence 10011101⋯ is generated by a recurrence relation of length 3: $x_{n+3} \equiv c_0 x_n + c_1 x_{n+1} + c_2 x_{n+2} \pmod 2$. Find the coefficients c_0, c_1, c_2.

3. The LFSR sequence 100100011110⋯ is generated by a recurrence relation of length 4: $x_{n+4} \equiv c_0 x_n + c_1 x_{n+1} + c_2 x_{n+2} + c_3 x_{n+3} \pmod 2$. Find the coefficients c_0, c_1, c_2, c_3.

4. The LFSR sequence 10111001⋯ is generated by a recurrence of length 3: $x_{n+3} \equiv c_0 x_n + c_1 x_{n+1} + c_2 x_{n+2} \pmod 2$. Find the coefficients c_0, c_1, and c_2.

5. Suppose we build an LFSR machine that works mod 3 instead of mod 2. It uses a recurrence of length 2 of the form

   $$x_{n+2} \equiv c_0 x_n + c_1 x_{n+1} \pmod 3$$

   to generate the sequence 1, 1, 0, 2, 2, 0, 1, 1. Set up

and solve the matrix equation to find the coefficients c0 and c1.

6. The sequence
   x1=0, x2=1, x3=1, x4=1, x5=0, x6=0, x7=1, x8=0, x9=1, ⋯
   is generated by a recurrence relation

   $$x_{n+3} \equiv c_0 x_n + c_1 x_{n+1} + c_2 x_{n+2} \pmod 2.$$

   Determine x10, x11, x12.

7. Consider the sequence starting k1=1, k2=0, k3=1 and defined by the length 3 recurrence $k_{n+3} \equiv k_n + k_{n+1} + k_{n+2} \pmod 2$. This sequence can also be given by a length 2 recurrence. Determine this length 2 recurrence by setting up and solving the appropriate matrix equations.

8. Suppose we build an LFSR-type machine that works mod 2. It uses a recurrence of length 2 of the form

   $$x_{n+2} \equiv c_0 x_n + c_1 x_{n+1} + 1 \pmod 2$$

   to generate the sequence 1,1,0,0,1,1,0,0. Find c0 and c1.

9. Suppose you modify the LFSR method to work mod 5 and you use a (not quite linear) recurrence relation

   $$x_{n+2} \equiv c_0 x_n + c_1 x_{n+1} + 2 \pmod 5, \quad x1=0,\ x2=1,\ x3=1,$$

x4=0.

Find the coefficients c0 and c1.

10.

1. Suppose you make a modified LFSR machine using the recurrence relation $x_{n+2} \equiv A+Bx_n+Cx_{n+1} \pmod 2$, where A, B, C are constants. Suppose the output starts 0, 1, 1, 0, 0, 1, 1, 0, 0. Find the constants A, B, C.

2. Show that the sequence does not satisfy any linear recurrence of the form $x_{n+2} \equiv c_0x_n+c_1x_{n+1} \pmod 2$.

11. Bob has a great idea to generate pseudorandom bytes. He takes the decimal expansion of π, which we assume is random, and chooses three consecutive digits in this expansion, starting at a randomly chosen point. He then regards what he gets as a three-digit integer, which he writes as a 10-digit number in binary. Finally, he chooses the last eight binary digits to get a byte. For example, if the digits of π that he chooses are 159, he changes this to 0010011111. This yields the byte 10011111.

    1. Show that this pseudorandom number generator produces some bytes more

often than others.

2. Suppose Bob modifies his algorithm so that if his three-digit decimal integer is greater than or equal to 512, then he discards it and tries again. Show that this produces random output (assuming the same is true for π).

12. Suppose you have a Geiger counter and a radioactive source. If the Geiger counter counts an even number of radioactive particles in a second, you write 0. If it records an odd number of particles, you write 1. After a period of time, you have a binary sequence. It is reasonable to expect that the probability pn that n particles are counted in a second satisfies a Poisson distribution

    pn=e−λλnn! for n ≥ 0,

    where λ is a parameter (in fact, λ is the average number of particles per second).

    1. Show that if 0<λ<1 then p0>p1>p2⋯.

    2. Show that if λ<1 then the binary sequence you obtain is expected to have more 0s than 1s.

    3. More generally, show that, whenever

$\lambda \geq 0$,

$\text{Prob}(n \text{ is even}) = e^{-\lambda}\cosh(\lambda) \quad \text{Prob}(n \text{ is odd}) = e^{-\lambda}\sinh(\lambda)$.

4. Show that for every $\lambda \geq 0$,

   $\text{Prob}(n \text{ is even}) > \text{Prob}(n \text{ is odd})$.

   This problem shows that, although a Geiger counter might be a good source of randomness, the naive method of using it to obtain a pseudorandom sequence is biased.

13.

1. Suppose that during the PRGA of RC4, there occur values of $i=i_0$ and $j=j_0$ such that $j_0=i_0+1$ and $S[i_0+1]=1$. The next values of i, j in the algorithm are i_1, j_1, with $i_1=i+1$ and $j_1=j+1$. Show that $S[i_1+1]=1$, so this property continues for all future i, j.

2. The values of i, j before i_0, j_0 are $i -= i_0 - 1$ and $j -= j_0 - 1$. Show that $S[i-]=1$, so if this property occurs, then it occurred for all previous values of i, j.

3. The starting values of i and j are $i=0$ and $j=0$. Use this to show that there are never

values of i, j such that j=i+1 and S[i]=1.

14. Let f(x) be a one-way function. In Section 5.1, it was pointed out that usually the least significant bits of f(s+j) for j=1, 2, 3, … (s is a seed) can be used to give a pseudorandom sequence of bits. Show how to append some bits to f(x) to obtain a new one-way function for which the sequence of least significant bits is not pseudorandom.

5.5 Computer Problems

1. The following sequence was generated by a linear feedback shift register. Determine the recurrence that generated it.

```
1, 0, 1, 0, 0, 1, 1, 0, 1, 1,
0, 0, 0, 1, 0, 0, 1, 0, 0,
0, 0, 1, 1, 1, 0, 0, 0, 0, 0,
1, 0, 1, 1, 1, 1, 1, 1, 0,
0, 1, 0, 1, 0, 1, 0, 0, 0, 1,
1, 0, 0, 1, 1, 1, 1, 0, 1,
1, 1, 0, 1, 0, 1, 1, 0, 1, 0,
0, 1, 1, 0, 1, 1, 0, 0, 0,
1, 0, 0, 1, 0, 0, 0, 0, 1, 1,
1, 0, 0, 0, 0, 0, 1, 0, 1,
1, 1, 1, 1, 1
```

   (It is stored in the downloadable computer files (`bit.ly/2JbcS6p`) under the name *L101*.)

2. The following are the first 100 terms of an LFSR output. Find the coefficients of the recurrence.

```
1, 0, 0, 1, 1, 0, 0, 1, 0, 0,
1, 1, 1, 0, 0, 0, 1, 1, 0,
0, 0, 1, 0, 1, 0, 0, 0, 1, 1,
1, 1, 0, 1, 1, 0, 0, 1, 1,
1, 1, 1, 0, 1, 0, 1, 0, 1, 0,
0, 1, 0, 1, 1, 0, 1, 1, 0,
1, 0, 1, 1, 0, 0, 0, 0, 1, 1,
0, 1, 1, 1, 0, 0, 1, 0, 1,
0, 1, 1, 1, 1, 0, 0, 0, 0, 0,
0, 0, 1, 0, 0, 0, 1, 0, 0,
1, 0, 0, 0, 0
```

(It is stored in the downloadable computer files (`bit.ly/2JbcS6p`) under the name *L100*.)

3. The following ciphertext was obtained by XORing an LFSR output with the plaintext.

```
0, 1, 1, 0, 0, 0, 1, 0, 1, 0,
1, 1, 1, 0, 0, 1, 1, 1, 0,
1, 0, 1, 0, 0, 0, 1, 0, 0, 0,
1, 1, 0, 0, 0, 1, 0, 1, 0,
1, 1, 1, 0, 0, 1, 1, 1, 0, 1,
0, 1
```

Suppose you know the plaintext starts

```
```

```
1, 0, 0, 1, 0, 0, 1, 0, 0, 1,
0, 0, 1, 0, 0
```

Find the plaintext. (The ciphertext is stored in the downloadable computer files (`bit.ly/2JbcS6p`) under the name *L011*.)

Chapter 6 Block Ciphers

6.1 Block Ciphers

In many classical cryptosystems, changing one letter in the plaintext changes exactly one letter in the ciphertext. In the shift, affine, and substitution ciphers, a given letter in the ciphertext always comes from exactly one letter in the plaintext. This greatly facilitates finding the key using frequency analysis. In the Vigenère system, the use of blocks of letters, corresponding to the length of the key, makes the frequency analysis more difficult, but still possible, since there is

no interaction among the various letters in each block. Block ciphers avoid these problems by encrypting blocks of several letters or numbers simultaneously. A change of one character in a plaintext block should change potentially all the characters in the corresponding ciphertext block.

The Playfair cipher in Section 2.6 is a simple example of a block cipher, since it takes two-letter blocks and encrypts them to two-letter blocks. A change of one letter of a plaintext pair always changes at least one letter, and usually both letters, of the ciphertext pair. However, blocks of two letters are too small to be secure, and frequency analysis, for example, is usually

successful.

Many of the modern cryptosystems that will be treated later in this book are block ciphers. For example, DES operates on blocks of 64 bits. AES uses blocks of 128 bits. RSA sometimes uses blocks more than 1000 bits long, depending on the modulus used. All of these block lengths are long enough to be secure against attacks such as frequency analysis.

Claude Shannon, in one of the fundamental papers on the theoretical foundations of cryptography [Shannon1], gave two properties that a good cryptosystem should have in order to hinder statistical analysis: **diffusion** and **confusion**.

Diffusion means that if we change a character of the plaintext, then several characters of the ciphertext should change, and, similarly, if we change a character of the ciphertext, then several characters of the plaintext should change. This means that frequency statistics of letters, digrams, etc. in the plaintext are diffused over several characters in the ciphertext, which means that much more ciphertext is needed to do a meaningful statistical attack.

Confusion means that the key does not relate in a simple way to the ciphertext. In particular, each character of the ciphertext should depend on several parts of the key. When a situation like

this happens, the cryptanalyst probably needs to solve for the entire key simultaneously, rather than piece by piece.

The Vigenère and substitution ciphers do not have the properties of diffusion and confusion, which is why they are so susceptible to frequency analysis.

The concepts of diffusion and confusion play a role in any well-designed block cipher. Of course, a disadvantage (which is precisely the cryptographic advantage) of diffusion is error propagation: A small error in the ciphertext becomes a major error in the decrypted message, and usually means the decryption is unreadable.

The natural way of using a block cipher is to convert blocks of plaintext to blocks of ciphertext, independently and one at a time. This is called the electronic codebook (ECB) mode. Although it seems like the obvious way to implement a block cipher, we'll see that it is insecure and that there are much better ways to use a block cipher. For example, it is possible to use feedback from the blocks of ciphertext in the encryption of subsequent blocks of plaintext. This leads to the cipher block chaining (CBC) mode and cipher feedback (CFB) mode of operation. These are discussed in Section 6.3.

For an extensive discussion of block ciphers, see [Schneier].

6.2 Hill Ciphers

This section is not needed for understanding the rest of the chapter. It is included as an example of a block cipher.

In this section, we discuss the Hill cipher, which is a block cipher invented in 1929 by Lester Hill. It seems never to have been used much in practice. Its significance is that it was perhaps the first time that algebraic methods (linear algebra, modular arithmetic) were used in cryptography in an essential way. As we'll see in later chapters, algebraic methods now occupy a central position in the subject.

Choose an integer n, for example n=3. The key is an n×n matrix M whose entries are integers mod 26. For example, let

$$M=1234561198.$$

The message is written as a series of row vectors. For example, if the message is *abc*, we change this to the single row vector (0, 1, 2). To encrypt, multiply the vector by the matrix (traditionally, the matrix appears on the right in the multiplication; multiplying on the left would yield a similar theory) and reduce mod 26:

$$(0,\ 1,\ 2)\,(1234561198) \equiv (0,\ 23,\ 22) (\mathrm{mod}\ 26).$$

Therefore, the ciphertext is *AXW*. (The fact that the first letter a remained unchanged is a random occurrence; it is

not a defect of the method.)

In order to decrypt, we need the determinant of M to satisfy

gcd(det(M), 26)=1.

This means that there is a matrix N with integer entries such that
$MN \equiv I \pmod{26}$, where I is the n×n identity matrix.

In our example, det(M)=−3. The inverse of M is

−13−1411−334−256−1913−3.

Since 17 is the inverse of −3 mod 26, we replace −1/3 by 17 and reduce mod 26 to obtain

N=22516172415131.

The reader can check that
$MN \equiv I \pmod{26}$.

For more on finding inverses of matrices mod n, see Section 3.8. See also Example 15 in the Computer Appendices.

The decryption is accomplished by multiplying by N, as follows:

$$(0,\ 23,\ 22)\,(22516172415131) \equiv (0,\ 1,\ 2) \pmod{26}$$

In the general method with an n×n matrix, break the plaintext into blocks of n characters and change each block to a vector of n integers between 0 and 25 using a=0, b=1, …, z=25. For example, with the matrix M as above, suppose our plaintext is

blockcipher.

This becomes (we add an x to fill the last space)

111142102815741723.

Now multiply each vector by M, reduce the answer mod 26, and change back to letters:

(1, 11, 14)M=(199, 183, 181) ≡ (17, 1, 25)(mod 26)=RBZ(2, 10, 2)M=(64, 72, 82) ≡ (12, 20, 4)(mod 26)=MUE, etc.

In our case, the ciphertext is

RBZMUEPYONOM.

It is easy to see that changing one letter of plaintext will usually change n letters of ciphertext. For example, if block is changed to clock, the first three letters of ciphertext change from RBZ to SDC. This makes frequency counts less

effective, though they are not impossible when n is small. The frequencies of two-letter combinations, called **digrams**, and three-letter combinations, **trigrams**, have been computed. Beyond that, the number of combinations becomes too large (though tabulating the results for certain common combinations would not be difficult). Also, the frequencies of combinations are so low that it is hard to get meaningful data without a very large amount of text.

Now that we have the ciphertext, how do we decrypt? Simply break the ciphertext into blocks of length n, change each to a vector, and multiply on the right by the inverse matrix N. In our example, we have

RBZ=(17, 1, 25)↦(17, 1, 25)N=(755, 427, 66) ≡ (1, 11, 14)=blo,

and similarly for the remainder of the ciphertext.

For another example, see Example 21 in the Computer Appendices.

The Hill cipher is difficult to decrypt using only the ciphertext, but it succumbs easily to a known plaintext attack. If we do not know n, we can try various values until we find the right one. So suppose n is known. If we have n of the blocks of plaintext of size n, then we can use the plaintext and the corresponding ciphertext to obtain a matrix equation for M (or for N, which might be more useful). For example, suppose we know that n=2 and we have the plaintext

howareyoutoday=71422017424142019143024

corresponding to the ciphertext

ZWSENIUSPLJVEU=25221841382018151192142O

The first two blocks yield the matrix equation

714220abcd ≡ 2522184(mod 26).

Unfortunately, the matrix 714220 has determinant −308, which is not invertible mod 26 (though this matrix could be used to reduce greatly the number of choices for the encryption matrix). Therefore, we replace the last row of the equation, for example, by the fifth block to obtain

7142019abcd ≡ 25221511(mod 26).

In this case, the matrix 7142019 is invertible mod 26:

$$7142019 - 1 \equiv 5101821 \pmod{26}.$$

We obtain

$$M \equiv 510182125221511 \equiv 1512113 \pmod{26}.$$

Because the Hill cipher is vulnerable to this attack, it cannot be regarded as being very strong.

A chosen plaintext attack proceeds by the same strategy, but is a little faster. Again, if you do not know n, try various possibilities until one works. So suppose n is known. Choose the first block of plaintext to be baaa···=1000…, the second to be abaa···=0100…, and continue through the nth block being … aaab=…0001. The blocks of ciphertext will be the rows of the matrix M.

For a chosen ciphertext attack, use the

same strategy as for chosen plaintext, where the choices now represent ciphertext. The resulting plaintext will be the rows of the inverse matrix N.

6.3 Modes of Operation

Suppose we have a block cipher. It can encrypt a block of plaintext of a fixed size, for example 64 bits. There are many circumstances, however, where it is necessary to encrypt messages that are either longer or shorter than the cipher's block length. For example, a bank may be sending a terabyte of data to another bank. Or you might be sending a short message that needs to be encrypted one letter at a time since you want to produce ciphertext output as quickly as you write the plaintext input.

Block ciphers can be run in many different modes of operation, allowing

users to choose appropriate modes to meet the requirements of their applications. There are five common modes of operation: electronic codebook (ECB), cipher block chaining (CBC), cipher feedback (CFB), output feedback (OFB), and counter (CTR) modes. We now discuss these modes.

6.3.1 Electronic Codebook (ECB)

The natural manner for using a block cipher is to break a long piece of plaintext into appropriately sized blocks of plaintext and process each block separately with the encryption function EK. This is known as the electronic codebook (ECB) mode of operation. The plaintext P is broken into smaller

chunks P=[P1, P2, ⋯, PL] and the ciphertext is

C=[C1, C2, ⋯, CL]

where Cj=EK(Pj) is the encryption of Pj using the key K.

There is a natural weakness in the ECB mode of operation that becomes apparent when dealing with long pieces of plaintext. Say an adversary Eve has been observing communication between Alice and Bob for a long enough period of time. If Eve has managed to acquire some plaintext pieces corresponding to the ciphertext pieces that she has observed, she can start to build up a codebook with which she can decipher future communication between Alice and Bob. Eve never needs to calculate

the key K; she just looks up a ciphertext message in her codebook and uses the corresponding plaintext (if available) to decipher the message.

This can be a serious problem since many real-world messages consist of repeated fragments. E-mail is a prime example. An e-mail between Alice and Bob might start with the following header:

```
Date: Tue, 29 Feb 2000 13:44:38
-0500 (EST)
```

The ciphertext starts with the encrypted version of “Date: Tu”. If Eve finds that this piece of ciphertext often occurs on a Tuesday, she might be able to guess, without knowing any of the plaintext,

that such messages are e-mail sent on Tuesdays. With patience and ingenuity, Eve might be able to piece together enough of the message's header and trailer to figure out the context of the message. With even greater patience and computer memory, she might be able to piece together important pieces of the message.

Another problem that arises in ECB mode occurs when Eve tries to modify the encrypted message being sent to Bob. She might be able to extract important portions of the message and use her codebook to construct a false ciphertext message that she can insert in the data stream.

6.3.2 Cipher Block Chaining (CBC)

One method for reducing the problems that occur in ECB mode is to use chaining. Chaining is a feedback mechanism where the encryption of a block depends on the encryption of previous blocks.

In particular, encryption proceeds as

$$C_j = E_K(P_j \oplus C_{j-1}),$$

while decryption proceeds as

$$P_j = D_K(C_j) \oplus C_{j-1},$$

where C0 is some chosen initial value. As usual, EK and DK denote the encryption and decryption functions for the block cipher.

Thus, in CBC mode, the plaintext is XORed with the previous ciphertext block and the result is encrypted. Figure 6.1 depicts CBC.

Figure 6.1 Cipher Block Chaining Mode.

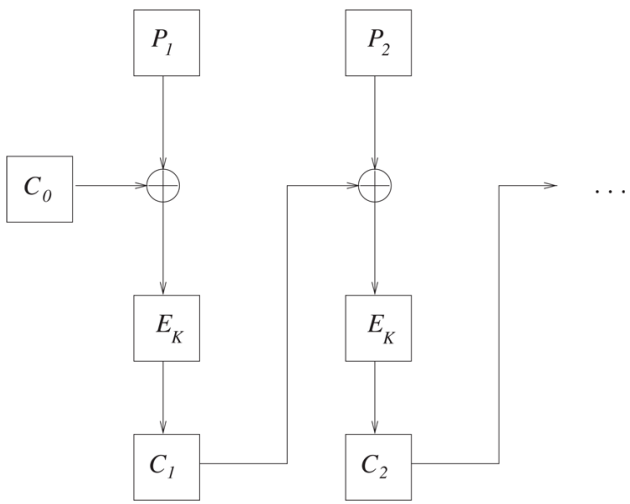


Figure 6.1 Full Alternative Text

The initial value C0 is often called the initialization vector, or the IV. If we use a fixed value for C0, say C0=0, and ever have the same plaintext message, the

result will be that the resulting ciphertexts will be the same. This is undesirable since it allows the adversary to deduce that the same plaintext was created. This can be very valuable information, and can often be used by the adversary to infer the meaning of the original plaintext.

In practice, this problem is handled by always choosing C0 randomly and sending C0 in the clear along with the first ciphertext C1. By doing so, even if the same plaintext message is sent repeatedly, an observer will see a different ciphertext each time.

6.3.3 Cipher Feedback (CFB)

One of the problems with both the CBC and ECB methods is that encryption (and hence decryption) cannot begin until a complete block of plaintext data is available. The cipher feedback mode operates in a manner that is very similar to the way in which LFSRs are used to encrypt plaintext. Rather than use linear recurrences to generate random bits, the cipher feedback mode is a stream mode of operation that produces pseudorandom bits using the block cipher EK. In general, CFB operates in a k-bit mode, where each application produces k random bits for XORing with the plaintext. For our discussion, however, we focus on the eight-bit version of CFB. Using the eight-bit CFB allows one 8-bit piece of message (e.g.,

a single character) to be encrypted without having to wait for an entire block of data to be available. This is useful in interactive computer communications, for example.

For concreteness, let's assume that our block cipher encrypts blocks of 64 bits and outputs blocks of 64 bits (the sizes of the registers can easily be adjusted for other block sizes). The plaintext is broken into 8-bit pieces: P=[P1, P2, …], where each Pj has eight bits, rather than the 64 bits used in ECB and CBC. Encryption proceeds as follows. An initial 64-bit X1 is chosen. Then for j=1, 2, 3, …, the following is performed:

$$O_j = L_8 E_K(X_j) \qquad C_j = P_j \oplus O_j \qquad X_{j+1} = R_{56}(X_j) \,\|\, C_j,$$

where L8(X) denotes the 8 leftmost bits

of X, R56(X) denotes the rightmost 56 bits of X, and X‖Y denotes the string obtained by writing X followed by Y. We present the CFB mode of operation in Figure 6.2.

Figure 6.2 Cipher Feedback Mode.

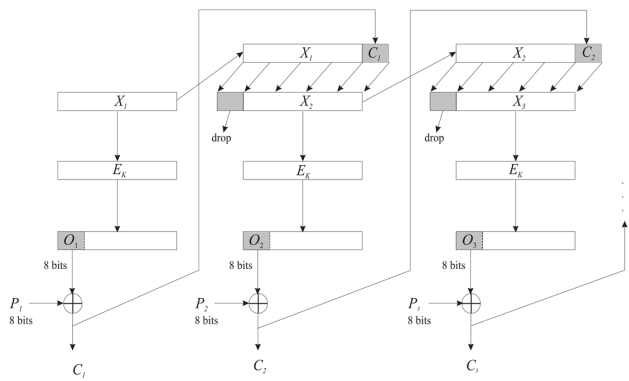


Figure 6.2 Full Alternative Text

Decryption is done with the following steps:

Pj=Cj⊕L8(EK(Xj))Xj+1=R56(Xj) ‖ Cj.

We note that decryption does not

involve the decryption function, DK. This would be an advantage of running a block cipher in a stream mode in a case where the decryption function for the block cipher is slower than the encryption function.

Let's step through one round of the CFB algorithm. First, we have a 64-bit register that is initialized with X1. These 64 bits are encrypted using EK and the leftmost eight bits of EK(X1) are extracted and XORed with the 8-bit P1 to form C1. Then C1 is sent to the recipient. Before working with P2, the 64-bit register X1 is updated by extracting the rightmost 56 bits. The eight bits of C1 are appended on the right to form X2=R56(X1)||C1. Then P2

is encrypted by the same process, but using X2 in place of X1. After P2 is encrypted to C2, the 64-bit register is updated to form

$$X3=R56(X2)\|C2=R48(X1)\|C1\|C2.$$

By the end of the 8th round, the initial X1 has disappeared from the 64-bit register and $X9=C1\|C2\|\cdots\|C8$. The Cj continue to pass through the register, so for example $X20=C12\|C13\|\cdots\|C19$.

Note that CFB encrypts the plaintext in a manner similar to one-time pads or LFSRs. The key K and the numbers Xj are used to produce binary strings that are XORed with the plaintext to produce the ciphertext. This is a much different type of encryption than the ECB and CBC, where the ciphertext is the output

of DES.

In practical applications, CFB is useful because it can recover from errors in transmission of the ciphertext. Suppose that the transmitter sends the ciphertext blocks C1, C2, …, Ck, …, and C1 is corrupted during transmission, so that the receiver observes C˜1, C2, …. Decryption takes C˜1 and produces a garbled version of P1 with bit errors in the locations that C˜1 had bit errors. Now, after decrypting this ciphertext block, the receiver forms an incorrect X2, which we denote X˜2. If X1 was (*, *, *, *, *, *, *, *), then X2˜=(*, *, *, *, *, *, *, C˜1). When the receiver gets an uncorrupted C2 and decrypts, then a completely garbled

version of P2 is produced. When forming X3, the decrypter actually forms X˜3=(*, *, *, *, *, *, C˜1, C2). The decrypter repeats this process, ultimately getting bad versions of P1, P2, ⋯, P9. When the decrypter calculates X9, the error block has moved to the leftmost block of X˜9 as X˜9=(C˜1, C2, ⋯, C8). At the next step, the error will have been flushed from the X10 register, and X10 and subsequent registers will be uncorrupted. For a simplified version of these ideas, see Exercise 18.

6.3.4 Output Feedback (OFB)

The CBC and CFB modes of operation exhibit a drawback in that errors

propagate for a duration of time corresponding to the block size of the cipher. The output feedback mode (OFB) is another example of a stream mode of operation for a block cipher where encryption is performed by XORing the message with a pseudorandom bit stream generated by the block cipher. One advantage of the OFB mode is that it avoids error propagation.

Much like CFB, OFB may work on chunks of different sizes. For our discussion, we focus on the eight-bit version of OFB, where OFB is used to encrypt eight-bit chunks of plaintext in a streaming mode. Just as for CFB, we break our plaintext into eight-bit pieces,

with P=P1, P2, ⋯, . We start with an initial value X1, which has a length equal to the block length of the cipher, for example, 64 bits (the sizes of the registers can easily be adjusted for other block sizes). X1 is often called the IV, and should be chosen to be random. X1 is encrypted using the key K to produce 64 bits of output, and the leftmost eight bits O1 of the ciphertext are extracted. These are then XORed with the first eight bits P1 of the plaintext to produce eight bits of ciphertext, C1.

So far, this is the same as what we were doing in CFB. But OFB differs from CFB in what happens next. In order to iterate, CFB updates the register X2 by extracting the right 56 bits of X1 and

appending C1 to the right side. Rather than use the ciphertext, OFB uses the output of the encryption. That is, OFB updates the register X2 by extracting the right 56 bits of X1 and appending O1 to the right side.

In general, the following procedure is performed for j=1, 2, 3, ⋯:

$$O_j = L_8E_K(X_j) \quad X_{j+1} = R_{56}(X_j) \| O_j \quad C_j = P_j \oplus O_j.$$

We depict the steps for the OFB mode of operation in Figure 6.3. Here, the output stream Oj is the encryption of the register containing the previous output from the block cipher. This output is then treated as a keystream and is XORed with the incoming plaintexts Pj to produce a stream of ciphertexts. Decryption is very simple. We get the

plaintext Pj by XORing the corresponding ciphertext Cj with the output keystream Oj. Again, just like CFB, we do not need the decryption function DK.

Figure 6.3 Output Feedback Mode.

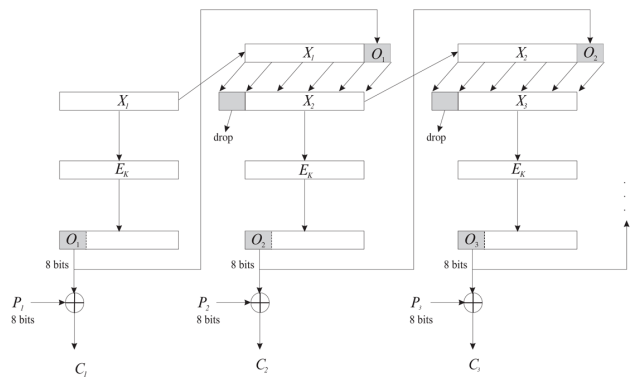


Figure 6.3 Full Alternative Text

So why would one want to build a stream cipher this way as opposed to the way the CFB stream cipher was built? There are a few key advantages to the

OFB strategy. First, the generation of the Oj output key stream may be performed completely without any plaintext. What this means is that the key stream can be generated in advance. This might be desirable for applications where we cannot afford to perform encryption operations as the plaintext message arrives.

Another advantage lies in its performance when errors are introduced to the ciphertext. Suppose a few errors are introduced to Cj when it is delivered to the receiver. Then only those corresponding bits in the plaintext are corrupted when decryption is performed. Since we build future output streams using the encryption of the

register, and not using the corrupted ciphertext, the output stream will always remain clean and the errors in the ciphertext will not propagate.

To summarize, CFB required the register to completely flush itself of errors, which produced an entire block length of garbled plaintext bits. OFB, on the other hand, will immediately correct itself.

There is one problem associated with OFB, however, that is common to all stream ciphers that are obtained by XORing pseudorandom numbers with plaintext. If Eve knows a particular plaintext Pj and ciphertext Cj, she can conduct the following attack. She first calculates

$$O_j = C_j \oplus P_j$$

to get out the key stream. She may then create any false plaintext P_j' she wants. Now, to produce a ciphertext, she merely has to XOR with the output stream she calculated:

$$C_j' = P_j' \oplus O_j.$$

This allows her to modify messages.

6.3.5 Counter (CTR)

The counter (CTR) mode builds upon the ideas that were used in the OFB mode. Just like OFB, CTR creates an output key stream that is XORed with chunks of plaintext to produce ciphertext. The main difference between CTR and OFB lies in the fact that the output stream Oj in CTR is not linked to

previous output streams.

CTR starts with the plaintext broken into eight-bit pieces, P=P1, P2, ⋯. We begin with an initial value X1, which has a length equal to the block length of the cipher, for example, 64 bits. Now, X1 is encrypted using the key K to produce 64 bits of output, and the leftmost eight bits of the ciphertext are extracted and XORed with P1 to produce eight bits of ciphertext, C1.

Now, rather than update the register X2 to contain the output of the block cipher, we simply take X2=X1+1. In this way, X2 does not depend on previous output. CTR then creates new output stream by encrypting X2. Similarly, we may proceed by using X3=X2+1, and so on.

The jth ciphertext is produced by XORing the left eight bits from the encryption of the jth register with the corresponding plaintext Pj.

In general, the procedure for CTR is

$$Xj=Xj-1+1Oj=L8EK(Xj)Cj=Pj\oplus Oj$$

for j=2, 3, ⋯, and is presented in Figure 6.4. The reader might wonder what happens to Xj if we continually add 1 to it. Shouldn't it eventually become too large? This is unlikely to happen, but if it does, we simply wrap around and start back at 0.

Figure 6.4 Counter Mode.

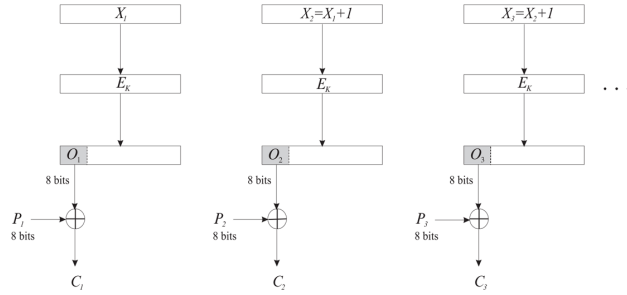


Figure 6.4 Full Alternative Text

Just like OFB, the registers Xj can be calculated ahead of time, and the actual encryption of plaintext is simple in that it involves just the XOR operation. As a result, its performance is identical to OFB's when errors are introduced in the ciphertext. The advantage of CTR mode compared to OFB, however, stems from the fact that many output chunks Oj may be calculated in parallel. We do not have to calculate Oj before calculating Oj+1. This makes CTR mode ideal for parallelizing.

6.4 Multiple Encryption

As technology improves and more sophisticated attacks are developed, encryptions systems become less secure and need to be replaced. There are two main approaches to achieving increased security. The first involves using encryption multiple times and leads, for example, to triple encryption. The second approach is to find a new system that is more secure, a potentially lengthy process.

We start by describing the idea behind multiple encryption schemes. The idea is to encrypt the same plaintext multiple times using the same algorithm with

different keys. **Double encryption** encrypts the plaintext by first encrypting with one key and then encrypting again using another key. For example, if the keyspace for single encryption has 56 bits, hence 256 keys, then the new keyspace consists of 2112 keys. One might guess that double encryption should therefore double the security. This, however, is not true. Merkle and Hellman showed that the double encryption scheme actually has the security level of a 57-bit key. The reduction from 2112 to 257 makes use of the **meet-in-the-middle attack**, which is described in the next section.

Since double encryption has a weakness, **triple encryption** is often used. This

appears to have a level of security approximately equivalent to a 112-bit key (when the single encryption has a 56-bit key). There are at least two ways that triple encryption can be implemented. One is to choose three keys, K1, K2, K3, and perform EK1(EK2(EK3(m))). This type of triple encryption is sometimes called *EEE*. The other is to choose two keys, K1 and K2, and perform EK1(DK2(EK1(m))). This is sometimes called *EDE*. When K1=K2, this reduces to single encryption. Therefore, a triple encryption machine that is communicating with an older machine that still uses single encryption can simply set K1=K2 and proceed. This compatibility is the reason for using

DK2 instead of EK2 in the middle; the use of D instead of E gives no extra cryptographic strength. Both versions of triple encryption are resistant to meet-in-the-middle attacks (compare with Exercise 11). However, there are other attacks on the two-key version ([Merkle-Hellman] and [van Oorschot-Wiener]) that indicate possible weaknesses, though they require so much memory as to be impractical.

Another strengthening of encryption was proposed by Rivest. Choose three keys, K1, K2, K3, and perform $K3 \oplus EK2(K1 \oplus m)$. In other words, modify the plaintext by *XOR*ing with K1, then apply encryption with K2, then *XOR* the result with K3. This method,

when used with DES, is known as DESX and has been shown to be fairly secure. See [Kilian-Rogaway].

6.5 Meet-in-the-Middle Attacks

Alice and Bob are using an encryption method. The encryption functions are called EK, and the decryption functions are called DK, where K is a key. We assume that if someone knows K, then she also knows EK and DK (so Alice and Bob could be using one of the classical, nonpublic key systems such as DES or AES). They have a great idea. Instead of encrypting once, they use two keys K1 and K2 and encrypt twice. Starting with a plaintext message m, the ciphertext is c=EK2(EK1(m)). To decrypt, simply compute m=DK1(DK2(c)). Eve will need to

discover both K1 and K2 to decrypt their messages.

Does this provide greater security? For many cryptosystems, applying two encryptions is the same as using an encryption for some other key. For example, the composition of two affine functions is still an affine function (see Exercise 11 in Chapter 2). Similarly, using two RSA encryptions (with the same n) with exponents e1 and e2 corresponds to doing a single encryption with exponent e1e2. In these cases, double encryption offers no advantage. However, there are systems, such as DES (see Subsection 7.4.1) where the composition of two encryptions is not simply encryption with another key. For

these, double encryption might seem to offer a much higher level of security. However, the following attack shows that this is not really the case, as long as we have a computer with a lot of memory.

Assume Eve has intercepted a message m and a doubly encrypted ciphertext c=EK2(EK1(m)). She wants to find K1 and K2. She first computes two lists:

1. EK(m) for all possible keys K
2. DL(c) for all possible keys L.

Finally, she compares the two lists and looks for matches. There will be at least one match, since the correct pair of keys will be one of them, but it is likely that there will be many matches. If there are

several matches, she then takes another plaintext–ciphertext pair and determines which of the pairs (K, L) she has found will encrypt the plaintext to the ciphertext. This should greatly reduce the list. If there is still more than one pair remaining, she continues until only one pair remains (or she decides that two or more pairs give the same double encryption function). Eve now has the desired pair K1, K2.

If Eve has only one plaintext–ciphertext pair, she still has reduced the set of possible key pairs to a short list. If she intercepts a future transmission, she can try each of these possibilities and obtain a very short list of meaningful plaintexts.

If there are N possible keys, Eve needs to compute N values EL(m). She then needs to compute N numbers DL(c) and compare them with the stored list. But these 2N computations (plus the comparisons) are much less than the N2 computations required for searching through all key pairs K1, K2.

This meet-in-the-middle procedure takes slightly longer than the exhaustive search through all keys for single encryption. It also takes a lot of memory to store the first list. However, the conclusion is that double encryption does not significantly raise the level of security.

Similarly, we could use triple encryption, using triples of keys. A

similar attack brings the level of security down to at most what one might naively expect from double encryption, namely squaring the possible number of keys.

Example

Suppose the single encryption has 256 possible keys and the block cipher inputs and outputs blocks of 64 bits, as is the case with DES. The first list has 256 entries, each of which is a 64-bit block. The probability that a given block in List 1 matches a given block in List 2 is 2−64. Since there are 256 entries in List 2, we expect that a given block in List 1 matches 2562−64=2−8 entries of List 2. Running through the 256 elements in List 1, we expect 2562−8=248 pairs (K, L) for which

there are matches between List 1 and List 2.

We know that one of these 248 matches is from the correct pair (K1, K2), and the other matches are probably caused by randomness. If we take a random pair (K, L) and try it on a new plaintext–ciphertext (m1, c1), then EL(EK(m1)) is a 64-bit block that has probability 2−64 of matching the 64-bit block c1. Therefore, among the approximately 248 random pairs, we expect 2482−64=2−16 matches between EL(EK(m1)) and c1. In other words, it is likely that the second plaintext–ciphertext pair eliminates all extraneous solutions and leaves only the correct key pair (K1, K2). If not, a third round

should complete the task.

6.6 Exercises

1. The ciphertext *YIFZMA* was encrypted by a Hill cipher with matrix 91323. Find the plaintext.

2. The matrix 1111 mod 26 is not suitable for the matrix in a Hill cipher. Why?

3. The ciphertext text *GEZXDS* was encrypted by a Hill cipher with a 2×2 matrix. The plaintext is *solved*. Find the encryption matrix M.

4. Consider the following combination of Hill and Vigenère ciphers: The key consists of three 2×2 matrices, M1, M2, M3. The plaintext letters are represented as integers mod 26. The first two are encrypted by M1, the next two by M2, the 5th and 6th by M3. This is repeated cyclically, as in the Vigenère cipher. Explain how to do a chosen plaintext attack on this system. Assume that you know that three 2×2 matrices are being used. State explicitly what plaintexts you would use and how you would use the outputs.

5. Eve captures Bob's Hill cipher machine, which uses a 2-by-2 matrix M mod 26. She tries a chosen

plaintext attack. She finds that the plaintext ba encrypts to HC and the plaintext zz encrypts to GT. What is the matrix M.

6. Alice uses a Hill cipher with a 3×3 matrix M that is invertible mod 26. Describe a chosen plaintext attack that will yield the entries of the matrix M. Explicitly say what plaintexts you will use.

7.
    1. The ciphertext text *ELNI* was encrypted by a Hill cipher with a 2×2 matrix. The plaintext is *dont*. Find the encryption matrix.

    2. Suppose the ciphertext is *ELNK* and the plaintext is still *dont*. Find the encryption matrix. Note that the second column of the matrix is changed. This shows that the entire second column of the encryption matrix is involved in obtaining the last character of the ciphertext.

8. Suppose the matrix 1234 is used for an encryption matrix in a Hill cipher. Find two plaintexts that encrypt to the same ciphertext.

9. Let a, b, c, d, e, f be integers mod 26. Consider the following combination of the Hill and affine

ciphers: Represent a block of plaintext as a pair (x, y) mod 26. The corresponding ciphertext (u, v) is

$$xyabcd+ef \equiv uv(\text{mod } 26).$$

Describe how to carry out a chosen plaintext attack on this system (with the goal of finding the key a, b, c, d, e, f). You should state explicitly what plaintexts you choose and how to recover the key.

10. Alice is sending a message to Bob using a Hill cipher with a 2×2 matrix. In fact, Alice is bored and her plaintext consists of the letter a repeated a few hundred times. Eve knows what system is being used, but not the key, and intercepts the ciphertext. State how Eve will recognize that the plaintext is one repeated letter and decide whether or not Eve can deduce the letter and/or the key. (Note: The solution very much depends on the fact that the repeated letter is a, rather than b, c, ….)

11. Let EK denote encryption (for some cryptosystem) with key K. Suppose that there are 235 possible keys K. Alice decides to encrypt a message m as follows:

    She chooses two keys K and L and double encrypts by computing

    $$c=EK(EK(EL(m)))$$

to get the ciphertext c. Suppose Eve knows Alice's method of encryption (but not K and L) and has at least two plaintext–ciphertext pairs. Describe a method that is *guaranteed* to yield the correct K and L (and maybe a *very small* additional set of incorrect pairs). Be explicit enough to say why you are using at least two plaintext–ciphertext pairs. Eve may do up to 250 computations.

12. Alice and Bob are arguing about which method of multiple encryption they should use. Alice wants to choose keys K1 and K2 and triple encrypt a message m as c=EK1(EK2(EK1(m))). Bob wants to encrypt m as c=EK1(EK1(EK2(EK2(m)))). Which method is more secure? Describe in detail an attack on the weaker encryption method.

13. Alice and Bob are trying to implement triple encryption. Let EK denote DES encryption with key K and let DK denote decryption.

    1. Alice chooses two keys, K1 and K2, and encrypts using the formula c=EK1(DK2(EK1(m))). Bob chooses two keys, K1 and K2, and encrypts using the formula c=EK1(EK2(EK2(m))). One of these methods is more secure than the other. Say which one is weaker and explicitly give the steps that can be used to attack the weaker system. You may

assume that you know ten plaintext–ciphertext pairs.

2. What is the advantage of using DK2 instead of EK2 in Alice's version?

14. Suppose E1 and E2 are two encryption methods. Let K1 and K2 be keys and consider the double encryption

    EK1, K2(m)=EK11(EK22(m)).

    1. Suppose you know a plaintext–ciphertext pair. Show how to perform a meet-in-the-middle attack on this double encryption.

    2. An affine encryption given by $x \mapsto \alpha x + \beta \pmod{26}$ can be regarded as a double encryption, where one encryption is multiplying the plaintext by α and the other is a shift by β. Assume that you have a plaintext and ciphertext that are long enough that α and β are unique. Show that the meet-in-the-middle attack from part (a) takes at most 38 steps (not including the comparisons between the lists). Note that this is much faster than a brute force search through all 312 keys.

15. Let EK denote DES encryption with key K. Suppose there is a public database Y consisting of 1010 DES keys and there is another public database Z of 1010 binary strings of length 64. Alice has five messages m1, m2, …, m5. She chooses a key K from Y and a string B from Z. She encrypts each message m by computing $c=EK(m)\oplus B$. She uses the same K and B for each of the messages. She shows the five plaintext–ciphertext pairs (mi, ci) to Eve and challenges Eve to find K and B. Alice knows that Eve's computer can do only 1015 calculations, and there are 1020 pairs (K, B), so Alice thinks that Eve cannot find the correct pair. However, Eve has taken a crypto course. Show how she can find the K and B that Alice used. You must state explicitly what Eve does. Statements such as "Eve makes a list" are not sufficient; you must include what is on the lists and how long they are.

16. Alice wants to encrypt her messages securely, but she can afford only an encryption machine that uses a 25-bit key. To increase security, she chooses 4 keys K1, K2, K3, K4 and encrypts four times:

    c=EK1(EK2(EK3(EK4(m)))).

    Eve finds several plaintext–ciphertext pairs (m, c) encrypted with this set of keys. Describe how she can find (with high probability) the keys

K1, K2, K3, K4. (For this problem, assume that Eve can do at most 260 computations, so she cannot try all 2100 combinations of keys.) (Note: If you use only one of the plaintext–ciphertext pairs in your solution, you probably have not done enough to determine the keys.)

17. Show that the decryption procedures given for the CBC and CFB modes actually perform the desired decryptions.

18. Consider the following simplified version of the CFB mode. The plaintext is broken into 32-bit pieces: P=[P1, P2, …], where each Pj has 32 bits, rather than the eight bits used in CFB. Encryption proceeds as follows. An initial 64-bit X1 is chosen. Then for j=1, 2, 3, …, the following is performed:

    $$C_j = P_j \oplus L32(E_K(X_j)) \quad X_{j+1} = R32(X_j) \| C_j,$$

    where L32(X) denotes the 32 leftmost bits of X, R32(X) denotes the rightmost 32 bits of X, and X‖Y denotes the string obtained by writing X followed by Y.

    1. Find the decryption algorithm.

    2. The ciphertext consists of 32-bit blocks C1, C2, C3, C4, …. Suppose that a transmission error causes C1 to be received as $\tilde{C}1 \neq C1$, but that

C2, C3, C4, … are received correctly. This corrupted ciphertext is then decrypted to yield plaintext blocks P˜1, P˜2, …. Show that P˜1 ≠ P1, but that P˜i=Pi for all i ≥ 4. Therefore, the error affects only three blocks of the decryption.

19. The cipher block chaining (CBC) mode has the property that it recovers from errors in ciphertext blocks. Show that if an error occurs in the transmission of a block Cj, but all the other blocks are transmitted correctly, then this affects only two blocks of the decryption. Which two blocks?

20. In CTR mode, the initial X1 has 64 bits and is sent unencrypted to the receiver. (a) If X1 is chosen randomly every time a message is encrypted, approximately how many messages must be sent in order for there to be a good chance that two messages use the same X1 ? (b) What could go wrong if the same X1 is used for two different messages? (Assume that the key K is not changed.)

21. Suppose that in CBC mode, the final plaintext block Pn is incomplete; that is, its length M is less than the usual block size of, say, 64 bits. Often, this last block is padded with a binary string to make it have full length. Another method that can

be used is called **ciphertext stealing**, as follows:

1. Compute $Yn-1=EK(Cn-2 \oplus Pn-1)$.
2. Compute $Cn=LM(Yn-1)$, where LM means we take the leftmost M bits.
3. Compute $Cn-1=EK(Pn||064-M) \oplus Yn-1$, where $Pn||064-M$ denotes Pn with enough 0s appended to give it the length of a full 64-bit block.
4. The ciphertext is $C1C2 \cdots Cn-1Cn$. Therefore, the ciphertext has the same length as the plaintext.

Suppose you receive a message that used this ciphertext stealing for the final blocks (the ciphertext blocks C1, …Cn−2 were computed in the usual way for CBC). Show how to decrypt the ciphertext (you have the same key as the sender).

22. Suppose Alice has a block cipher with 250 keys, Bob has one with 240 keys, and Carla has one with 230 keys. The only known way to break single encryption with each system is by brute force, namely trying all keys. Alice uses her system with single encryption. But Bob uses his with double encryption, and Carla uses hers with triple encryption. Who has the most secure system? Who

has the weakest? (Assume that double and triple encryption do not reduce to using single or double encryption, respectively. Also, assume that some plaintext-ciphertext pairs are available for Alice's single encryption, Bob's double encryption, and Carla's triple encryption.)

6.7 Computer Problems

1. The following is the ciphertext of a Hill cipher

```
zirkzwopjjoptfapuhfhadrq
```

using the matrix

12344321112462964.

Decrypt.

Chapter 7 The Data Encryption Standard

7.1 Introduction

In 1973, the National Bureau of Standards (NBS), later to become the National Institute of Standards and Technology (NIST), issued a public request seeking a cryptographic algorithm to become a national standard. IBM submitted an algorithm called LUCIFER in 1974. The NBS forwarded it to the National Security Agency, which reviewed it and, after some modifications, returned a version that was essentially the Data Encryption

Standard (DES) algorithm. In 1975, NBS released DES, as well as a free license for its use, and in 1977 NBS made it the official data encryption standard.

DES was used extensively in electronic commerce, for example in the banking industry. If two banks wanted to exchange data, they first used a public key method such as RSA to transmit a key for DES, then they used DES for transmitting the data. It had the advantage of being very fast and reasonably secure.

From 1975 on, there was controversy surrounding DES. Some regarded the key size as too small. Many were worried about NSA's involvement. For

example, had they arranged for it to have a “trapdoor" – in other words, a secret weakness that would allow only them to break the system? It was also suggested that NSA modified the design to avoid the possibility that IBM had inserted a trapdoor in LUCIFER. In any case, the design decisions remained a mystery for many years.

In 1990, Eli Biham and Adi Shamir showed how their method of differential cryptanalysis could be used to attack DES, and soon thereafter they showed how these methods could succeed faster than brute force. This indicated that perhaps the designers of DES had been aware of this type of attack. A few years later, IBM released some details of the

design criteria, which showed that indeed they had constructed the system to be resistant to differential cryptanalysis. This cleared up at least some of the mystery surrounding the algorithm.

DES lasted for a long time, but became outdated. Brute force searches (see Section 7.5), though expensive, can now break the system. Therefore, NIST replaced it with the system AES (see Chapter 8) in the year 2000. However, it is worth studying DES since it represents a popular class of algorithms and it was one of the most frequently used cryptographic algorithms in history.

DES is a block cipher; namely, it breaks

the plaintext into blocks of 64 bits, and encrypts each block separately. The actual mechanics of how this is done is often called a **Feistel system**, after Horst Feistel, who was part of the IBM team that developed LUCIFER. In the next section, we give a simple algorithm that has many of the characteristics of this type of system, but is small enough to use as an example. In Section 7.3, we show how differential cryptanalysis can be used to attack this simple system. We give the DES algorithm in Section 7.4, Finally, in Section 7.5, we describe some methods used to break DES.

7.2 A Simplified DES-Type Algorithm

The DES algorithm is rather unwieldy to use for examples, so in the present section we present an algorithm that has many of the same features, but is much smaller. Like DES, the present algorithm is a block cipher. Since the blocks are encrypted separately, we assume throughout the present discussion that the full message consists of only one block.

The message has 12 bits and is written in the form L0R0, where L0 consists of the first six bits and R0 consists of the last six bits. The key K has nine bits.

The ith round of the algorithm transforms an input Li−1Ri−1 to the output LiRi using an eight-bit key Ki derived from K.

The main part of the encryption process is a function f(Ri−1, Ki) that takes a six-bit input Ri−1 and an eight-bit input Ki and produces a six-bit output. This will be described later.

The output for the ith round is defined as follows:

$$L_i = R_{i-1} \text{ and } R_i = L_{i-1} \oplus f(R_{i-1}, K_i),$$

where $\oplus$ denotes XOR, namely bit-by-bit addition mod 2. This is depicted in Figure 7.1.

Figure 7.1 One Round of a

Figure 7.1 Full Alternative Text

This operation is performed for a certain number of rounds, say n, and produces the ciphertext LnRn.

How do we decrypt? Start with LnRn and switch left and right to obtain RnLn. (Note: *This switch is built into the DES*

encryption algorithm, so it is not needed when decrypting DES.) Now use the same procedure as before, but with the keys Ki used in reverse order Kn, …, K1. Let's see how this works. The first step takes RnLn and gives the output

[Ln] [Rn⊕f(Ln, Kn)].

We know from the encryption procedure that Ln=Rn−1 and Rn=Ln−1⊕f(Rn−1, Kn). Therefore,

[Ln] [Rn⊕f(Ln, Kn)]=[Rn−1] [Ln−1⊕f(Rn−1, Kn)⊕f(Ln, Kn)]=[Rn−1] [Ln−1].

The last equality again uses Ln=Rn−1, so that f(Rn−1, Kn)⊕f(Ln, Kn) is 0. Similarly, the second step of decryption sends Rn−1Ln−1 to Rn−2Ln−2. Continuing, we see that the decryption

process leads us back to R0L0. Switching the left and right halves, we obtain the original plaintext L0R0, as desired.

Note that the decryption process is essentially the same as the encryption process. We simply need to switch left and right and use the keys Ki in reverse order. Therefore, both the sender and receiver use a common key and they can use identical machines (though the receiver needs to reverse left and right inputs).

So far, we have said nothing about the function f. In fact, any f would work in the above procedures. But some choices of f yield much better security than others. The type of f used in DES is

similar to that which we describe next. It is built up from a few components.

The first function is an expander. It takes an input of six bits and outputs eight bits. The one we use is given in Figure 7.2.

Figure 7.2 The Expander Function

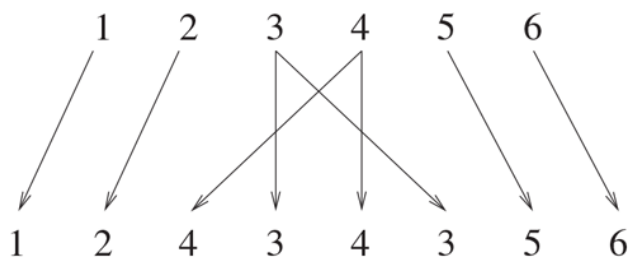


Figure 7.2 Full Alternative Text

This means that the first input bit yields the first output bit, the third input bit yields both the fourth and the sixth output bits, etc. For example, 011001 is

expanded to 01010101.

The main components are called S-boxes. We use two:

S1 [10101000111001110011100000110011001000001111010 11]
S2 [10000011010111100101101010101100011111001000110 0].

The input for an S-box has four bits. The first bit specifies which row will be used: 0 for the first row, 1 for the second. The other three bits represent a binary number that specifies the column: 000 for the first column, 001 for the second, ..., 111 for the last column. The output for the S-box consists of the three bits in the specified location. For example, an input of 1010 for S1 means we look at the second row, third column, which yields the output 110.

The key K consists of nine bits. The key Ki for the ith round of encryption is obtained by using eight bits of K, starting with the ith bit. For example, if K=010011001, then K4=01100101 (after five bits, we reached the end of K, so the last two bits were obtained from the beginning of K).

We can now describe f(Ri−1, Ki). The input Ri−1 consists of six bits. The expander function is used to expand it to eight bits. The result is XORed with Ki to produce another eight-bit number. The first four bits are sent to S1, and the last four bits are sent to S2. Each S-box outputs three bits, which are concatenated to form a six-bit number. This is f(Ri−1, Ki). We present this in

Figure 7.3.

Figure 7.3 The Function f(Ri –1, Ki)

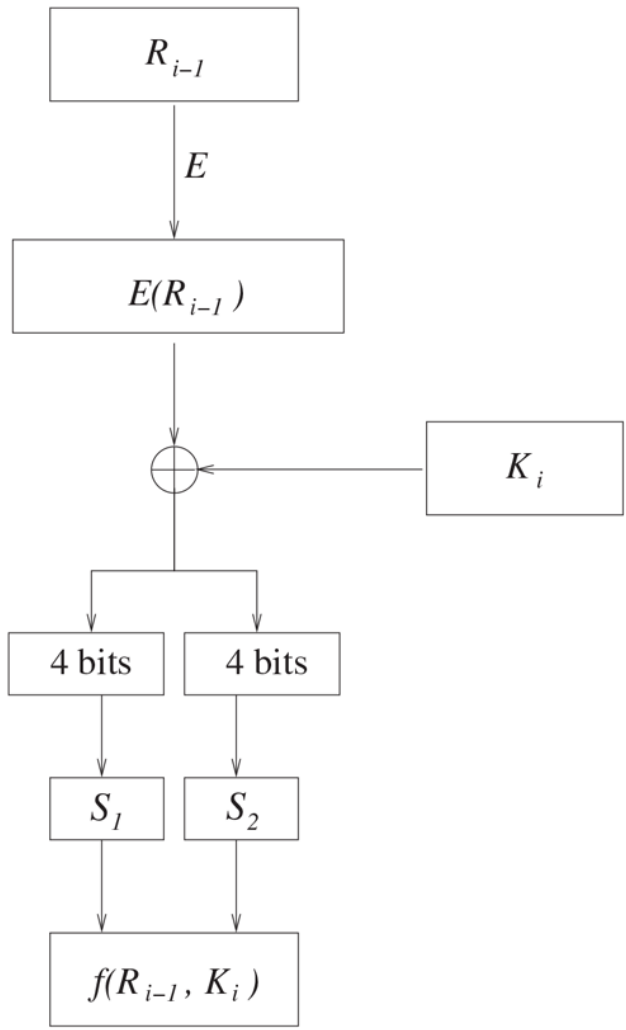


Figure 7.3 Full Alternative Text

For example, suppose Ri−1=100110 and Ki=01100101. We have

$$E(100110) \oplus Ki = 10101010 \oplus 01100101 = 11001111.$$

The first four bits are sent to S1 and the last four bits are sent to S2. The second row, fifth column of S1 contains 000. The second row, last column of S2 contains 100. Putting these outputs one after the other yields f(Ri −1, Ki)=000100.

We can now describe what happens in one round. Suppose the input is

$$Li{-}1Ri{-}1 = 011100100110$$

and Ki=01100101, as previously. This means that Ri−1=100110, as in the example just discussed. Therefore, f(Ri −1, Ki)=000100. This is XORed with Li

−1=011100 to yield Ri=011000. Since Li=Ri−1, we obtain

LiRi=100110011000.

The output becomes the input for the next round.

For work on this and another simplified DES algorithm and how they behave under multiple encryption, see [Konikoff-Toplosky].

7.3 Differential Cryptanalysis

This section is rather technical and can be skipped on a first reading.

Differential cryptanalysis was introduced by Biham and Shamir around 1990, though it was probably known much earlier to the designers of DES at IBM and NSA. The idea is to compare the differences in the ciphertexts for suitably chosen pairs of plaintexts and thereby deduce information about the key. Note that the difference of two strings of bits can be found by XORing them. Because the key is introduced by XORing with E(Ri−1), looking at the

XOR of the inputs removes the effect of the key at this stage and hence removes some of the randomness introduced by the key. We'll see that this allows us to deduce information as to what the key could be.

7.3.1 Differential Cryptanalysis for Three Rounds

We eventually want to describe how to attack the above system when it uses four rounds, but we need to start by analyzing three rounds. Therefore, we temporarily start with L1R1 instead of L0R0.

The situation is now as follows. We have obtained access to a three-round

encryption device that uses the preceding procedure. We know all the inner workings of the encryption algorithm such as the S-boxes, but we do not know the key. We want to find the key by a chosen plaintext attack. We use various inputs L1R1 and obtain outputs L4R4.

We have

$$R2=L1\oplus f(R1, K2)L3=R2=L1\oplus f(R1, K2)R4=L3\oplus f(R3, K4)=L1\oplus f(R1, K2)$$

Suppose we have another message $L1*R1*$ with $R1=R1*$. For each i, let $Ri'=Ri\oplus Ri*$ and $Li'=Li\oplus Li*$. Then $Li'Ri'$ is the "difference" (or sum; we are working mod 2) of LiRi and $Li*Ri*$. The preceding calculation applied to $L1*R1*$ yields a formula for $R4*$. Since we have assumed that $R1=R1*$, we have

f(R1, K2)=f(R1*, K2). Therefore, f(R1, K2)⊕f(R1*, K2)=0 and

$$R4' = R4 \oplus R4^* = L1' \oplus f(R3, K4) \oplus f(R3^*, K4).$$

This may be rearranged to

$$R4' \oplus L1' = f(R3, K4) \oplus f(R3^*, K4).$$

Finally, since R3=L4 and R3*=L4*, we obtain

$$R4' \oplus L1' = f(L4, K4) \oplus f(L4^*, K4).$$

Note that if we know the input XOR, namely L1′R1′, and if we know the outputs L4R4 and L4*R4*, then we know everything in this last equation except K4.

Now let’s analyze the inputs to the S-boxes used to calculate f(L4, K4) and f(L4*, K4). If we start with L4, we first

expand and then XOR with K4 to obtain $E(L4) \oplus K4$, which are the bits sent to S1 and S2. Similarly, $L4*$ yields $E(L4*) \oplus K4$. The XOR of these is

$$E(L4) \oplus E(L4*) = E(L4 \oplus L4*) = E(L)4'$$

(the first equality follows easily from the bit-by-bit description of the expansion function). Therefore, we know

1. the XORs of the inputs to the two S-boxes (namely, the first four and the last four bits of $E(L4')$);

2. the XORs of the two outputs (namely, the first three and the last three bits of $R4' \oplus L1'$).

Let's restrict our attention to S1. The analysis for S2 will be similar. It is fairly fast to run through all pairs of four-bit inputs with a given XOR (there are only 16 of them) and see which ones

give a desired output XOR. These can be computed once for all and stored in a table.

For example, suppose we have input XOR equal to 1011 and we are looking for output XOR equal to 100. We can run through the input pairs (1011, 0000), (1010, 0001), (1001, 0010), ..., each of which has XOR equal to 1011, and look at the output XORs. We find that the pairs (1010, 0001) and (0001, 1010) both produce output XORs 100. For example, 1010 means we look at the second row, third column of S1, which is 110. Moreover, 0001 means we look at the first row, second column, which is 010. The output XOR is therefore $110 \oplus 010 = 100$.

We know L4 and L4*. For example, suppose L4=101110 and L4*=000010. Therefore, E(L4)=10111110 and E(L4*)=00000010, so the inputs to S1 are $1011 \oplus$K4L and $0000 \oplus$K4L, where K4L denotes the left four bits of K4. If we know that the output XOR for S1 is 100, then ($1011 \oplus$K4L, $0000 \oplus$K4L) must be one of the pairs on the list we just calculated, namely (1010, 0001) and (0001, 1010). This means that K4L=0001 or 1010.

If we repeat this procedure a few more times, we should be able to eliminate one of the two choices for K4 and hence determine four bits of K. Similarly, using S2, we find four more bits of K. We therefore know eight of the nine bits

of K. The last bit can be found by trying both possibilities and seeing which one produces the same encryptions as the machine we are attacking.

Here is a summary of the procedure (for notational convenience, we describe it with both S-boxes used simultaneously, though in the examples we work with the S-boxes separately):

1. Look at the list of pairs with input XOR=E(L4′) and output XOR=R4′⊕L1′.
2. The pair (E(L4)⊕K4, E(L4∗)⊕K4) is on this list.
3. Deduce the possibilities for K4.
4. Repeat until only one possibility for K4 remains.

Example

We start with

L1R1=000111011011

and the machine encrypts in three rounds using the key K=001001101, though we do not yet know K. We obtain (note that since we are starting with L1R1, we start with the shifted key K2=01001101)

L4R4=000011100101.

If we start with

L1*R1*=101110011011

(note that R1=R1*), then

L4*R4*=100100011000.

We have E(L4)=00000011 and E(L4*)=10101000. The inputs to S1 have XOR equal to 1010 and the inputs to S2 have XOR equal to 1011. The S-boxes have output XOR

$R4' \oplus L1'1 = 111101 \oplus 101001 = 010100$, so the output XOR from S1 is 010 and that from S2 is 100.

For the pairs
(1001, 0011), (0011, 1001), S1 produces output XOR equal to 010. Since the first member of one of these pairs should be the left four bits of $E(L4) \oplus K4 = 0000 \oplus K4$, the first four bits of K4 are in {1001, 0011}. For the pairs (1100, 0111), (0111, 1100), S2 produces output XOR equal to 100. Since the first member of one of these pairs should be the right four bits of $E(L4) \oplus K4 = 0011 \oplus K4$, the last four bits of K4 are in {1111, 0100}.

Now repeat (with the same machine and the same key K) and with

L1R1=010111011011 and L1*R1*=101110011011.

A similar analysis shows that the first four bits of K4 are in {0011, 1000} and the last four bits are in {0100, 1011}. Combining this with the previous information, we see that the first four bits of K4 are 0011 and the last four bits are 0100. Therefore, K=00*001101 (recall that K4 starts with the fourth bit of K.

It remains to find the third bit of K. If we use K=000001101, it encrypts L1R1 to 001011101010, which is not L4R4, while K=001001101 yields the correct encryption. Therefore, the key is K=001001101.

7.3.2 Differential

Cryptanalysis for Four Rounds

Suppose now that we have obtained access to a four-round device. Again, we know all the inner workings of the algorithm except the key, and we want to determine the key. The analysis we used for three rounds still applies, but to extend it to four rounds we need to use more probabilistic techniques.

There is a weakness in the box S1. If we look at the 16 input pairs with XOR equal to 0011, we discover that 12 of them have output XOR equal to 011. Of course, we expect on the average that two pairs should yield a given output XOR, so the present case is rather extreme. A little variation is to be

expected; we'll see that this large variation makes it easy to find the key.

There is a similar weakness in S2, though not quite as extreme. Among the 16 input pairs with XOR equal to 1100, there are eight with output XOR equal to 010.

Suppose now that we start with randomly chosen R0 and R0* such that $R0' = R0 \oplus R0^* = 001100$. This is expanded to E(001100)=00111100. Therefore the input XOR for S1 is 0011 and the input XOR for S2 is 1100. With probability 12/16 the output XOR for S1 will be 011, and with probability 8/16 the output XOR for S2 will be 010. If we assume the outputs of the two S-boxes are independent, we see that the

combined output XOR will be 011010 with probability (12/16)(8/16) = 3/8. Because the expansion function sends bits 3 and 4 to both S1 and S2, the two boxes cannot be assumed to have independent outputs, but 3/8 should still be a reasonable estimate for what happens.

Now suppose we choose L0 and L0* so that $L0' = L0 \oplus L0^* = 011010$. Recall that in the encryption algorithm the output of the S-boxes is XORed with L0 to obtain R1. Suppose the output XOR of the S-boxes is 011010. Then $R1' = 011010 \oplus L0' = 000000$. Since $R1' = R1 \oplus R1^*$, it follows that R1=R1*.

Putting everything together, we see that if we start with two randomly chosen

messages with XOR equal to L0′R0′=011010001100, then there is a probability of around 3/8 that L1′R1′=001100000000.

Here's the strategy for finding the key. Try several randomly chosen pairs of inputs with XOR equal to 011010001100. Look at the outputs L4R4 and L4*R4*. Assume that L1′R1′=001100000000. Then use three-round differential cryptanalysis with L1′=001100 and the known outputs to deduce a set of possible keys K4. When L1′R1′=001100000000, which should happen around 3/8 of the time, this list of keys will contain K4, along with some other random keys. The remaining 5/8 of the time, the list should contain

random keys. Since there seems to be no reason that any incorrect key should appear frequently, the correct key K4 will probably appear in the lists of keys more often than the other keys.

Here is an example. Suppose we are attacking a four-round device. We try one hundred random pairs of inputs L0R0 and L0*R0*=L0R0⊕011010001100. The frequencies of possible keys we obtain are in the following table. We find it easier to look at the first four bits and the last four bits of K4 separately.

First four bits	Frequency	First four bits	Frequency
0000	12	1000	33
0001	7	1001	40
0010	8	1010	35
0011	15	1011	35
0100	4	1100	59
0101	3	1101	32
0110	4	1110	28
0111	6	1111	39

Last four bits	Frequency	Last four bits	Frequency
0000	14	1000	8
0001	6	1001	16
0010	42	1010	8
0011	10	1011	18
0100	27	1100	8
0101	10	1101	23
0110	8	1110	6
0111	11	1111	17

7.3-1 Full Alternative Text

It is therefore likely that K4=11000010. Therefore, the key K is 10*110000.

To determine the remaining bit, we proceed as before. We can compute that 000000000000 is encrypted to 100011001011 using K=101110000 and is encrypted to 001011011010 using K=100110000. If the machine we are attacking encrypts 000000000000 to 100011001011, we conclude that the

second key cannot be correct, so the correct key is probably K=101110000.

The preceding attack can be extended to more rounds by extensions of these methods. It might be noticed that we could have obtained the key at least as quickly by simply running through all possibilities for the key. That is certainly true in this simple model. However, in more elaborate systems such as DES, differential cryptanalytic techniques are much more efficient than exhaustive searching through all keys, at least until the number of rounds becomes fairly large. In particular, the reason that DES uses 16 rounds appears to be because differential cryptanalysis is more efficient than exhaustive search

until 16 rounds are used.

There is another attack on DES, called **linear cryptanalysis**, that was developed by Mitsuru Matsui [Matsui]. The main ingredient is an approximation of DES by a linear function of the input bits. It is theoretically faster than an exhaustive search for the key and requires around 243 plaintext–ciphertext pairs to find the key. It seems that the designers of DES had not anticipated linear cryptanalysis. For details of the method, see [Matsui].

7.4 DES

A block of plaintext consists of 64 bits. The key has 56 bits, but is expressed as a 64-bit string. The 8th, 16th, 24th, ..., bits are parity bits, arranged so that each block of eight bits has an odd number of 1s. This is for error detection purposes. The output of the encryption is a 64-bit ciphertext.

The DES algorithm, depicted in Figure 7.4, starts with a plaintext m of 64 bits, and consists of three stages:

1. The bits of m are permuted by a fixed initial permutation to obtain m0=IP(m). Write m0=L0R0, where L0 is the first 32 bits of m0 and R0 is the last 32 bits.

2. For $1 \leq i \leq 16$, perform the following:

   $$L_i = R_{i-1} \quad R_i = L_{i-1} \oplus f(R_{i-1}, K_i),$$

   where K_i is a string of 48 bits obtained from the key K and f is a function to be described later.

3. Switch left and right to obtain $R_{16}L_{16}$, then apply the inverse of the initial permutation to get the ciphertext $c = IP^{-1}(R_{16}L_{16})$.

Figure 7.4 The DES Algorithm

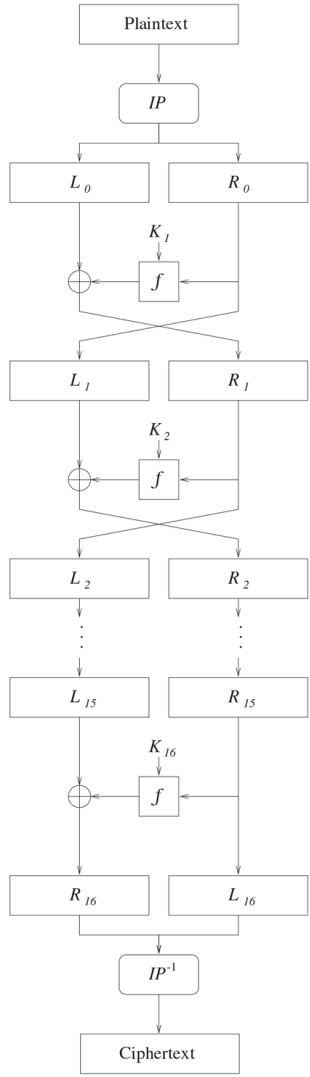


Figure 7.4 Full Alternative Text

Decryption is performed by exactly the same procedure, except that the keys K1, … , K16 are used in reverse order.

The reason this works is the same as for the simplified system described in Section 7.2. Note that the left–right switch in step 3 of the DES algorithm means that we do not have to do the left–right switch that was needed for decryption in Section 7.2.

We now describe the steps in more detail.

The initial permutation, which seems to have no cryptographic significance, but which was perhaps designed to make the algorithm load more efficiently into chips that were available in 1970s, can be described by the Initial Permutation table. This means that the 58th bit of m becomes the first bit of m0, the 50th bit of m becomes the second bit of m0, etc.

Initial Permutation															
58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5	63	55	47	39	31	23	15	7

7.4-2 Full Alternative Text

The function f(R, Ki), depicted in Figure 7.5, is described in several steps.

1. First, R is expanded to E(R) by the following table.

Expansion Permutation											
32	1	2	3	4	5	4	5	6	7	8	9
8	9	10	11	12	13	12	13	14	15	16	17
16	17	18	19	20	21	20	21	22	23	24	25
24	25	26	27	28	29	28	29	30	31	32	1

7.4-3 Full Alternative Text

   This means that the first bit of E(R) is the 32nd bit of R, etc. Note that E(R) has 48 bits.

2. Compute E(R)⊕Ki, which has 48 bits, and write it as B1B2···B8, where each Bj has six bits.

3. There are eight S-boxes S1, …, S8, given on page 150. Bj is the input for Sj. Write Bj=b1b2···b6. The row of the S-box is specified by b1b6 while b2b3b4b5 determines the column. For example, if B3=001001, we look at the row 01, which is the second row (00 gives the first row) and column 0100, which is the 5th column (0100 represents 4 in binary; the first column is numbered 0, so the

fifth is labeled 4). The entry in S3 in this location is 3, which is 3 in binary. Therefore, the output of S3 is 0011 in this case. In this way, we obtain eight four-bit outputs C1, C2, …C8.

4. The string C1C2···C8 is permuted according to the following table.

16	7	20	21	29	12	28	17	1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9	19	13	30	6	22	11	4	25

7.4-4 Full Alternative Text

The resulting 32-bit string is f(R, Kj).

Figure 7.5 The DES Function f(Ri–1, Ki)

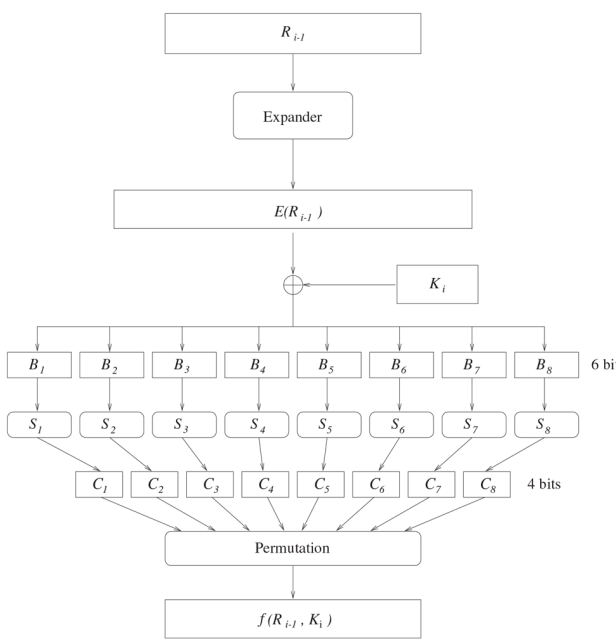


Figure 7.5 Full Alternative Text

Finally, we describe how to obtain K1, …K16. Recall that we start with a 64-bit K.

1. The parity bits are discarded and the remaining bits are permuted by the following table.

Key Permutation													
57	49	41	33	25	17	9	1	58	50	42	34	26	18
10	2	59	51	43	35	27	19	11	3	60	52	44	36
63	55	47	39	31	23	15	7	62	54	46	38	30	22
14	6	61	53	45	37	29	21	13	5	28	20	12	4

7.4-5 Full Alternative Text

Write the result as C0D0, where C0 and D0 have 28 bits.

2. For $1 \le i \le 16$, let $Ci=LSi(Ci-1)$ and $Di=LSi(Di-1)$. Here LSi means shift the input one or two places to the left, according to the following table.

Number of Key Bits Shifted per Round															
Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Shift	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

7.4-6 Full Alternative Text

3. 48 bits are chosen from the 56-bit string CiDi according to the following table. The output is Ki.

14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

7.4-7 Full Alternative Text

It turns out that each bit of the key is used in approximately 14 of the 16 rounds.

A few remarks are in order. In a good cipher system, each bit of the ciphertext should depend on all bits of the plaintext. The expansion E(R) is designed so that this will happen in only a few rounds. The purpose of the initial permutation is not completely clear. It has no cryptographic purpose. The S-boxes are the heart of the algorithm and provide the security. Their design was somewhat of a mystery until IBM published the following criteria in the early 1990s (for details, see [Coppersmith1]).

1. Each S-box has six input bits and four output bits.

This was the largest that could be put on one chip in 1974.

2. The outputs of the S-boxes should not be close to being linear functions of the inputs (linearity would have made the system much easier to analyze).

3. Each row of an S-box contains all numbers from 0 to 15.

4. If two inputs to an S-box differ by one bit, the outputs must differ by at least two bits.

5. If two inputs to an S-box differ in exactly the middle two bits, then the outputs differ in at least two bits.

6. If two inputs to an S-box differ in their first two bits but have the same last two bits, the outputs must be unequal.

7. There are 32 pairs of inputs having a given XOR. For each of these pairs, compute the XOR of the outputs. No more than eight of these output XORs should be the same. This is clearly to avoid an attack via differential cryptanalysis.

8. A criterion similar to (7), but involving three S-boxes.

In the early 1970s, it took several months of searching for a computer to find appropriate S-boxes. Now, such a search could be completed in a very short time.

7.4.1 DES Is Not a Group

One possible way of effectively increasing the key size of DES is to double encrypt. Choose keys K1 and K2 and encrypt a plaintext P by EK2(EK1(P)). Does this increase the security?

S-Boxes

S-box 1

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S-box 2

15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

S-box 3

10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

S-box 4

7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

S-box 5

2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

S-box 6

12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

S-box 7

4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

S-box 8

13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Full Alternative Text

Meet-in-the-middle attacks on cryptosystems are discussed in Section 6.5. It is pointed out that, if an attacker has sufficient memory, double encryption provides little extra protection. Moreover, if a cryptosystem

is such that double encryption is equivalent to a single encryption, then there is no additional security obtained by double encryption.

In addition, if double encryption is equivalent to single encryption, then the (single encryption) cryptosystem is much less secure than one might guess initially (see Exercise 3 in Chapter 12). If this were true for DES, for example, then an exhaustive search through all 256 keys could be replaced by a search of length around 228, which would be quite easy to do.

For affine ciphers (Section 2.2) and for RSA (Chapter 9), double encrypting with two keys K1 and K2 is equivalent to encrypting with a third key K3. Is the

same true for DES? Namely, is there a key K3 such that EK3=EK2EK1? This question is often rephrased in the equivalent form “Is DES a group?” (The reader who is unfamiliar with group theory can ask “Is DES closed under composition?”)

Fortunately, it turns out that DES is not a group. We sketch the proof. For more details, see [Campbell-Wiener]. Let E0 represent encryption with the key consisting entirely of 0s and let E1 represent encryption with the key consisting entirely of 1s. These keys are weak for cryptographic purposes (see Exercise 5). Moreover, D. Coppersmith found that applying E1∘E0 repeatedly to certain plaintexts yielded the original

plaintext after around 232 iterations. A sequence of encryptions (for some plaintext P)

E1E0(P), E1E0(E1E0(P)), E1E0(E1E0(E1E0(P))), …, (E1E0)n(P)=P,

where n is the smallest positive integer such that (E1E0)n(P)=P, is called a cycle of length n.

Lemma

If m is the smallest positive integer such that (E1E0)m(P)=P for all P, and n is the length of a cycle (so (E1E0)n(P0)=P0 for a particular P0), then n divides m.

Proof. Divide n into m, with remainder r. This means that m=nq+r for some integer q, and $0 \leq r < n$. Since (E1E0)n(P0)=P0, encrypting q times

with (E1E0)n leaves P0 unchanged. Therefore,

P0=(E1E0)m(P0)=(E1E0)r(E1E0)nq(P0)=(E1E0)r(P0).

Since n is the smallest positive integer such that (E1E0)n(P0)=P0, and $0 \le r<n$, we must have r=0. This means that m=nq, so n divides m.

Suppose now that DES is closed under composition. Then E1E0=EK for some key K. Moreover, EK2, EK3, … are also represented by DES keys. Since there are only 256 possible keys, we must have EKj=EKi for some integers i, j with $0 \le i<j \le 256$ (otherwise we would have 256+1 distinct encryption keys). Decrypt i times: EKj −i=DKiEKj=DKiEKi, which is the identity map. Since $0<j-i \le 256$, the

smallest positive integer m such that EKm is the identity map also satisfies $m \leq 256$.

Coppersmith found the lengths of the cycles for 33 plaintexts P0. By the lemma, m is a multiple of these cycle lengths. Therefore, m is greater than or equal to the least common multiple of these cycle lengths, which turned out to be around 10277. But if DES is closed under composition, we showed that $m \leq 256$. Therefore, DES is not closed under composition.

7.5 Breaking DES

DES was the standard cryptographic system for the last 20 years of the twentieth century, but, in the latter half of this period, DES was showing signs of age. In this section we discuss the breaking of DES.

From 1975 onward, there were questions regarding the strength of DES. Many in the academic community complained about the size of the DES keys, claiming that a 56-bit key was insufficient for security. In fact, a few months after the NBS release of DES, Whitfield Diffie and Martin Hellman published a paper titled “Exhaustive

cryptanalysis of the NBS Data Encryption Standard" [Diffie-Hellman2] in which they estimated that a machine could be built for $20 million (in 1977 dollars) that would crack DES in roughly a day. This machine's purpose was specifically to attack DES, which is a point that we will come back to later.

In 1987 DES came under its second five-year review. At this time, NBS asked for suggestions whether to accept the standard for another period, to modify the standard, or to dissolve the standard altogether. The discussions regarding DES saw NSA opposing the recertification of DES. The NBS argued at that time that DES was beginning to show signs of weakness, given the

current of level of computing power, and proposed doing away with DES entirely and replacing it with a set of NSA-designed algorithms whose inner workings would be known only to NSA and be well protected from reverse engineering techniques. This proposal was turned down, partially due to the fact that several key American industries would be left unprotected while replacement algorithms were put in place. In the end, DES was reapproved as a standard, yet in the process it was acknowledged that DES was showing signs of weakness.

Five years later, after NBS had been renamed NIST, the next five-year review came around. Despite the

weaknesses mentioned in 1987 and the technology advances that had taken place in five years, NIST recertified the DES algorithm in 1992.

In 1993, Michael Wiener, a researcher at Bell-Northern Research, proposed and designed a device that would attack DES more efficiently than ever before. The idea was to use the already well-developed switching technology available to the telephone industry.

The year 1996 saw the formulation of three basic approaches for attacking symmetric ciphers such as DES. The first method was to do distributive computation across a vast collection of machines. This had the advantage that it was relatively cheap, and the cost that

was involved could be easily distributed over many people. Another approach was to design custom architecture (such as Michael Wiener's idea) for attacking DES. This promised to be more effective, yet also more expensive, and could be considered as the high-end approach. The middle-of-the-line approach involved programmable logic arrays and has received the least attention to date.

In all three of these cases, the most popular approach to attacking DES was to perform an exhaustive search of the keyspace. For DES this seemed to be reasonable since, as mentioned earlier, more complicated cryptanalytic techniques had failed to show significant

improvement over exhaustive search.

The distributive computing approach to breaking DES became very popular, especially with the growing popularity of the Internet. In 1997 the RSA Data Security company issued a challenge to find the key and crack a DES encrypted message. Whoever cracked the message would win a $10,000 prize. Only five months after the announcement of the 1997 DES Challenge, Rocke Verser submitted the winning DES key. What is important about this is that it represents an example where the distributive computing approach had successfully attacked DES. Rocke Verser had implemented a program where thousands of computers spread

over the Internet had managed to crack the DES cipher. People volunteered time on their personal (and corporate) machines, running Verser's program under the agreement that Verser would split the winnings 60% to 40% with the owner of the computer that actually found the key. The key was finally found by Michael Sanders. Roughly 25% of the DES keyspace had been searched by that time. The DES Challenge phrase decrypted to "Strong cryptography makes the world a safer place."

In the following year, RSA Data Security issued DES Challenge II. This time the correct key was found by Distributed Computing Technologies,

and the message decrypted to “Many hands make light work.” The key was found after searching roughly 85% of the possible keys and was done in 39 days. The fact that the winner of the second challenge searched more of the keyspace and performed the task quicker than the first task shows the dramatic effect that a year of advancement in technology can have on cryptanalysis.

In the summer of 1998 the Electronic Frontier Foundation (EFF) developed a project called DES Cracker whose purpose was to reveal the vulnerability of the DES algorithm when confronted with a specialized architecture. The DES Cracker project was founded on a

simple principle: The average computer is ill suited for the task of cracking DES. This is a reasonable statement since ordinary computers, by their very nature, are multipurpose machines that are designed to handle generic tasks such as running an operating system or even playing a computer game or two. What the EFF team proposed to do was build specialized hardware that would take advantage of the parallelizable nature of the exhaustive search. The team had a budget of $200,000.

We now describe briefly the architecture that the EFF team's research produced. For more information regarding the EFF Cracker as well as the other tasks their cracker was designed to handle, see

[Gilmore].

The EFF DES Cracker consisted of basically three main parts: a personal computer, software, and a large collection of specialized chips. The computer was connected to the array of chips and the software oversaw the tasking of each chip. For the most part, the software didn't interact much with the hardware; it just gave the chips the necessary information to start processing and waited until the chips returned candidate keys. In this sense, the hardware efficiently eliminated a large number of invalid keys and only returned keys that were potentially promising. The software then processed each of the promising candidate keys on

its own, checking to see if one of the promising keys was in fact the actual key.

The DES Cracker took a 128-bit (16-byte) sample of ciphertext and broke it into two 64-bit (8-byte) blocks of text. Each chip in the EFF DES Cracker consisted of 24 search units. A search unit was a subset of a chip whose task was to take a key and two 64-bit blocks of ciphertext and attempt to decrypt the first 64-bit block using the key. If the "decrypted" ciphertext looked interesting, then the search unit decrypted the second block and checked to see if that "decrypted" ciphertext was also interesting. If both decrypted texts were interesting then the search unit told

the software that the key it checked was promising. If, when the first 64-bit block of ciphertext was decrypted, the decrypted text did not seem interesting enough, then the search unit incremented its key by 1 to form a new key. It then tried this new key, again checking to see if the result was interesting, and proceeded this way as it searched through its allotted region of keyspace.

How did the EFF team define an "interesting" decrypted text? First they assumed that the plaintext satisfied some basic assumption, for example that it was written using letters, numbers, and punctuation. Since the data they were decrypting was text, they knew each

byte corresponded to an eight-bit character. Of the 256 possible values that an eight-bit character type represented, only 69 characters were interesting (the uppercase and lowercase alphabet, the numbers, the space, and a few punctuation marks). For a byte to be considered interesting, it had to contain one of these 69 characters, and hence had a 69/256 chance of being interesting. Approximating this ratio to 1/4, and assuming that the decrypted bytes are in fact independent, we see that the chance that an 8-byte block of decrypted text was interesting is 1/48=1/65536. Thus only 1/65536 of the keys it examined were considered promising.

This was not enough of a reduction. The software would still spend too much time searching false candidates. In order to narrow down the field of promising key candidates even further, it was necessary to use the second 8-byte block of text. This block was decrypted to see if the result was interesting. Assuming independence between the blocks, we get that only 1/416=1/655362 of the keys could be considered promising. This significantly reduced the amount of keyspace that the software had to examine.

Each chip consisted of 24 search units, and each search unit was given its own region of the keyspace that it was responsible for searching. A single 40-

MHz chip would have taken roughly 38 years to search the entire keyspace. To reduce further the amount of time needed to process the keys, the EFF team used 64 chips on a single circuit board, then 12 boards to each chassis, and finally two chassis were connected to the personal computer that oversaw the communication with the software.

The end result was that the DES Cracker consisted of about 1500 chips and could crack DES in roughly 4.5 days on average. The DES Cracker was by no means an optimum model for cracking DES. In particular, each of the chips that it used ran at 40 MHz, which is slow by modern standards. Newer models could certainly be produced in the future that

employ chips running at much faster clock cycles.

This development strongly indicated the need to replace DES. There were two main approaches to achieving increased security. The first used DES multiple times and led to the popular method called Triple DES or 3DES. Multiple encryption for block ciphers is discussed in Section 6.4.

The second approach was to find a new system that employs a larger key size than 56 bits. This led to AES, which is discussed in Chapter 8.

7.6 Password Security

When you log in to a computer and enter your password, the computer checks that your password belongs to you and then grants access. However, it would be quite dangerous to store the passwords in a file in the computer. Someone who obtains that file would then be able to open anyone's account. Making the file available only to the computer administrator might be one solution; but what happens if the administrator makes a copy of the file shortly before changing jobs? The solution is to encrypt the passwords before storing them.

Let f(x) be a **one-way function**. This means that it is easy to compute f(x), but it is very difficult to solve y=f(x) for x. A password x can then be stored as f(x), along with the user's name. When the user logs in, and enters the password x, the computer calculates f(x) and checks that it matches the value of f(x) corresponding to that user. An intruder who obtains the password file will have only the value of f(x) for each user. To log in to the account, the intruder needs to know x, which is hard to compute since f(x) is a one-way function.

In many systems, the encrypted passwords are stored in a public file. Therefore, anyone with access to the system can obtain this file. Assume the

function f(x) is known. Then all the words in a dictionary, and various modifications of these words (writing them backward, for example) can be fed into f(x). Comparing the results with the password file will often yield the passwords of several users.

This **dictionary attack** can be partially prevented by making the password file not publicly available, but there is still the problem of the departing (or fired) computer administrator. Therefore, other ways of making the information more secure are also needed.

Here is another interesting problem. It might seem desirable that f(x) can be computed very quickly. However, a slightly slower f(x) can slow down a

dictionary attack. But slowing down f(x) too much could also cause problems. If f(x) is designed to run in a tenth of a second on a very fast computer, it could take an unacceptable amount of time to log in on a slower computer. There doesn't seem to be a completely satisfactory way to resolve this.

One way to hinder a dictionary attack is with what is called **salt**. Each password is randomly padded with an additional 12 bits. These 12 bits are then used to modify the function f(x). The result is stored in the password file, along with the user's name and the values of the 12-bit salt. When a user enters a password x, the computer finds the value of the salt for this user in the file,

then uses it in the computation of the modified f(x), which is compared with the value stored in the file.

When salt is used and the words in the dictionary are fed into f(x), they need to be padded with each of the 212=4096 possible values of the salt. This slows down the computations considerably. Also, suppose an attacker stores the values of f(x) for all the words in the dictionary. This could be done in anticipation of attacking several different password files. With salt, the storage requirements increase dramatically, since each word needs to be stored 4096 times.

The main purpose of salt is to stop attacks that aim at finding a random

person's password. In particular, it makes the set of poorly chosen passwords somewhat more secure. Since many people use weak passwords, this is desirable. Salt does not slow down an attack against an individual password (except by preventing use of over-the-counter DES chips; see below). If Eve wants to find Bob's password and has access to the password file, she finds the value of the salt used for Bob and tries a dictionary attack, for example, using only this value of salt corresponding to Bob. If Bob's password is not in the dictionary, this will fail, and Eve may have to resort to an exhaustive search of all possible passwords.

In many Unix password schemes, the

one-way function was based on DES. The first eight characters of the password are converted to seven-bit ASCII (see Section 4.1). These 56 bits become a DES key. If the password is shorter than eight symbols, it is padded with zeros to obtain the 56 bits. The "plaintext" of all zeros is then encrypted using 25 rounds of DES with this key. The output is stored in the password file. The function

password → output

is believed to be one-way. Namely, we know the "ciphertext," which is the output, and the "plaintext," which is all zeros. Finding the key, which is the password, amounts to a known plaintext attack on DES, which is generally

assumed to be difficult.

In order to increase security, salt is added as follows. A random 12-bit number is generated as the salt. Recall that in DES, the expansion function E takes a 32-bit input R (the right side of the input for the round) and expands it to 48 bits E(R). If the first bit of the salt is 1, the first and 25th bits of E(R) are swapped. If the second bit of the salt is 1, the second and 26th bits of E(R) are swapped. This continues through the 12th bit of the salt. If it is 1, the 12th and 36th bits of E(R) are swapped. When a bit of the salt is 0, it causes no swap. If the salt is all zero, then no swaps occur and we are working with the usual DES. In this way, the salt

means that 4096 variations of DES are possible.

One advantage of using salt to modify DES is that someone cannot use high-speed DES chips to compute the one-way function when performing a dictionary attack. Instead, a chip would need to be designed that tries all 4096 modifications of DES caused by the salt; otherwise the attack could be performed with software, which is much slower.

Salt in any password scheme is regarded by many as a temporary measure. As storage space increases and computer speed improves, a factor of 4096 quickly fades, so eventually a new system must be developed.

For more on password protocols, see Section 12.6.

7.7 Exercises

1. Consider the following DES-like encryption method. Start with a message of 2n bits. Divide it into two blocks of length n (a left half and a right half): M0M1. The key K consists of k bits, for some integer k. There is a function f(K, M) that takes an input of k bits and n bits and gives an output of n bits. One round of encryption starts with a pair MjMj+1. The output is the pair Mj +1Mj+2, where

   $Mj+2=Mj \oplus f(K, Mj+1)$.

   ($\oplus$ means XOR, which is addition mod 2 on each bit). This is done for m rounds, so the ciphertext is MmMm+1.

   1. If you have a machine that does the m-round encryption just described, how would you use the same machine to decrypt the ciphertext MmMm+1 (using the same key K)? Prove that your decryption method works.

   2. Suppose K has n bits and

f(K, M)=K⊕M, and suppose the encryption process consists of m=2 rounds. If you know only a ciphertext, can you deduce the plaintext and the key? If you know a ciphertext and the corresponding plaintext, can you deduce the key? Justify your answers.

3. Suppose K has n bits and f(K, M)=K⊕M, and suppose the encryption process consists of m=3 rounds. Why is this system not secure?

2. Bud gets a budget 2-round Feistel system. It uses a 32-bit L, a 32-bit R, and a 32-bit key K. The function is f(R, K)=R⊕K, with the same key for each round. Moreover, to avoid transmission errors, he always uses a 32-bit message M and lets L0=R0=M. Eve does not know Bud's key, but she obtains the ciphertext for one of Bud's encryptions. Describe how Eve can obtain the plaintext M and the key K.

3. As described in Section 7.6, a way of storing passwords on a computer is to use DES with the password as the key to encrypt a fixed plaintext (usually 000 . . . 0). The ciphertext is then stored in the file. When you log in, the procedure is repeated and the ciphertexts are compared. Why is this

method more secure than the similar-sounding method of using the password as the plaintext and using a fixed key (for example, 000…0)?

4. Nelson produces budget encryption machines for people who cannot afford a full-scale version of DES. The encryption consists of one round of a Feistel system. The plaintext has 64 bits and is divided into a left half L and a right half R. The encryption uses a function f(R) that takes an input string of 32 bits and outputs a string of 32 bits. (There is no key; anyone naive enough to buy this system should not be trusted to choose a key.) The left half of the ciphertext is C0=R and the right half is C1=L⊕f(R). Suppose Alice uses one of these machines to encrypt and send a message to Bob. Bob has an identical machine. How does he use the machine to decrypt the ciphertext he receives? Show that this decryption works (do not quote results about Feistel systems; you are essentially justifying that a special case works).

5.
    1. Let K=111…111 be the DES key consisting of all 1s. Show that if EK(P)=C, then EK(C)=P, so encryption twice with this key returns the plaintext. (Hint: The round keys are sampled from K. Decryption uses these keys in reverse order.)

2. Find another key with the same property as K in part (a).

6. Alice uses quadruple DES encryption. To save time, she chooses two keys, K1 and K2, and encrypts via c=EK1(EK1(EK2(EK2(m)))). One day, Alice chooses K1 to be the key of all 1s and K2 to be the key of all 0s. Eve is planning to do a meet-in-the-middle attack, but after examining a few plaintext–ciphertext pairs, she decides that she does not need to carry out this attack. Why? (Hint: Look at Exercise 5.)

7. For a string of bits S, let S^{-} denote the complementary string obtained by changing all the 1s to 0s and all the 0s to 1s (equivalently, $S^{-}=S \oplus 11111\ldots$). Show that if the DES key K encrypts P to C, then K^{-} encrypts P^{-} to C^{-}. (Hint: This has nothing to do with the structure of the S-boxes. To do the problem, just work through the encryption algorithm and show that the input to the S-boxes is the same for both encryptions. A key point is that the expansion of C^{-} is the complementary string for the expansion of C.)

8. Suppose we modify the Feistel setup as follows. Divide the plaintext into three equal blocks: L0, M0, R0. Let the key for the ith round be Ki and let f be some function that produces the appropriate

size output. The ith round of encryption is given by

$$L_i = R_{i-1},\ M_i = L_{i-1},\ R_i = f(K_i, R_{i-1}) \oplus M_{i-1}.$$

This continues for n rounds. Consider the decryption algorithm that starts with the ciphertext An, Bn, Cn and uses the algorithm

$$A_{i-1} = B_i,\ B_{i-1} = f(K_i, A_i) \oplus C_i,\ C_{i-1} = A_i.$$

This continues for n rounds, down to A0, B0, C0. Show that Ai=Li, Bi=Mi, Ci=Ri for all i, so that the decryption algorithm returns the plaintext. (Remark: Note that the decryption algorithm is similar to the encryption algorithm, but cannot be implemented on the same machine as easily as in the case of the Feistel system.)

9. Suppose EK(M) is the DES encryption of a message M using the key K. We showed in Exercise 7 that DES has the complementation property, namely that if y=EK(M) then $\bar{y} = E_{\bar{K}}(\bar{M})$, where $\bar{M}$ is the bit complement of M. That is, the bitwise complement of the key and the plaintext result in the bitwise complement of the DES ciphertext. Explain how an adversary can use this property in a brute force, chosen plaintext attack to reduce the expected number of keys that would be tried from 255 to 254. (Hint: Consider a chosen plaintext set of (M1, C1) and ($\bar{M1}$, C2)).

7.8 Computer Problems

1. (For those who are comfortable with programming)

    1. Write a program that performs one round of the simplified DES-type algorithm presented in Section 7.2.

    2. Create a sample input bitstring, and a random key. Calculate the corresponding ciphertext when you perform one round, two rounds, three rounds, and four rounds of the Feistel structure using your implementation. Verify that the decryption procedure works in each case.

    3. Let EK(M) denote four-round encryption using the key K. By trying all 29 keys, show that there are no weak keys for this simplified DES-type algorithm. Recall that a weak key is one such that when we encrypt a plaintext twice we get back the plaintext. That is, a weak key K satisfies EK(EK(M))=M for every possible M. (Note: For each key K, you need to find

some M such that EK(EK(M)) ≠ M.)

4. Suppose you modify the encryption algorithm EK(M) to create a new encryption algorithm EK′(M) by swapping the left and right halves after the four Feistel rounds. Are there any weak keys for this algorithm?

2. Using your implementation of EK(M) from Computer Problem 1(b), implement the CBC mode of operation for this simplified DES-type algorithm.

    1. Create a plaintext message consisting of 48 bits, and show how it encrypts and decrypts using CBC.

    2. Suppose that you have two plaintexts that differ in the 14th bit. Show the effect that this has on the corresponding ciphertexts.

Chapter 8 The Advanced Encryption Standard: Rijndael

In 1997, the National Institute of Standards and Technology put out a call for candidates to replace DES. Among the requirements were that the new algorithm should allow key sizes of 128, 192, and 256 bits, it should operate on blocks of 128 input bits, and it should work on a variety of different hardware, for example, eight-bit processors that could be used in smart cards and the 32-bit architecture commonly used in personal computers. Speed and cryptographic strength were also important considerations. In 1998, the

cryptographic community was asked to comment on 15 candidate algorithms. Five finalists were chosen: MARS (from IBM), RC6 (from RSA Laboratories), Rijndael (from Joan Daemen and Vincent Rijmen), Serpent (from Ross Anderson, Eli Biham, and Lars Knudsen), and Twofish (from Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson). Eventually, Rijndael was chosen as the Advanced Encryption Standard. The other four algorithms are also very strong, and it is likely that they will used in many future cryptosystems.

As with other block ciphers, Rijndael can be used in several modes, for example, ECB, CBC, CFB, OFB, and

CTR (see Section 6.3).

Before proceeding to the algorithm, we answer a very basic question: How do you pronounce Rijndael? We quote from their Web page:

If you're Dutch, Flemish, Indonesian, Surinamer or South-African, it's pronounced like you think it should be. Otherwise, you could pronounce it like "Reign Dahl," "Rain Doll," "Rhine Dahl". We're not picky. As long as you make it sound different from "Region Deal."

8.1 The Basic Algorithm

Rijndael is designed for use with keys of lengths 128, 192, and 256 bits. For simplicity, we'll restrict to 128 bits. First, we give a brief outline of the algorithm, then describe the various components in more detail.

The algorithm consists of 10 rounds (when the key has 192 bits, 12 rounds are used, and when the key has 256 bits, 14 rounds are used). Each round has a round key, derived from the original key. There is also a 0th round key, which is the original key. A round starts with an input of 128 bits and produces an output of 128 bits.

There are four basic steps, called **layers**, that are used to form the rounds:

1. The SubBytes Transformation (SB): This nonlinear layer is for resistance to differential and linear cryptanalysis attacks.

2. The ShiftRows Transformation (SR): This linear mixing step causes diffusion of the bits over multiple rounds.

3. The MixColumns Transformation (MC): This layer has a purpose similar to ShiftRows.

4. AddRoundKey (ARK): The round key is *XOR*ed with the result of the above layer.

A round is then

→SubBytes→ShiftRows→MixColumns→AddRou

Putting everything together, we obtain the following (see also Figure 8.1):

Figure 8.1The AES-Rijndael

Algorithm

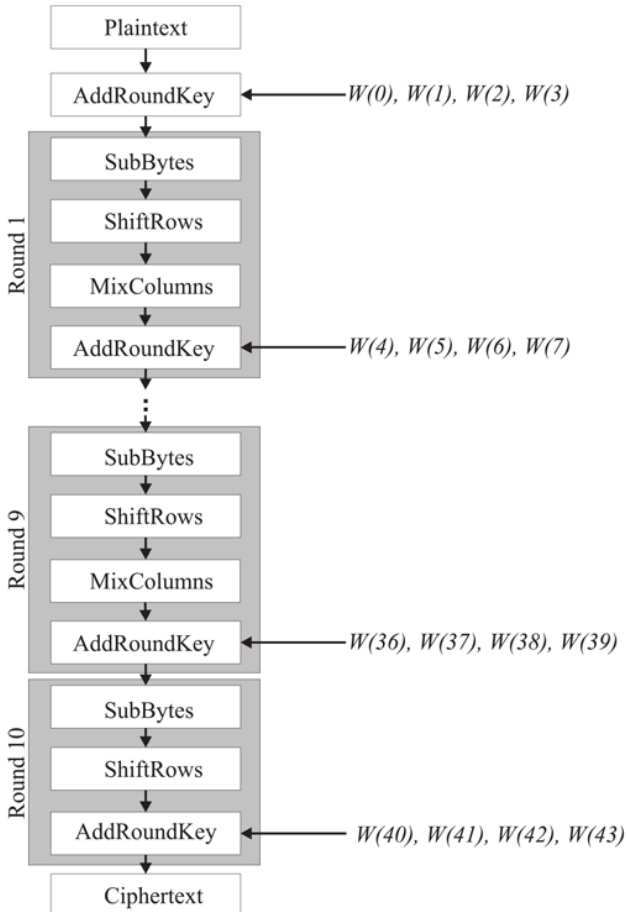


Figure 8.1 Full Alternative Text

Rijndael Encryption
1. ARK, using the 0th round key. 2. Nine rounds of SB, SR, MC, ARK, using round keys 1 to 9. 3. A final round: SB, SR, ARK, using the 10th round key.

8.1-1 Full Alternative Text

The final round uses the SubBytes,

ShiftRows, and AddRoundKey steps but omits MixColumns (this omission will be explained in the decryption section).

The 128-bit output is the ciphertext block.

8.2 The Layers

We now describe the steps in more detail. The 128 input bits are grouped into 16 bytes of eight bits each, call them

$$a_{0,0},\ a_{1,0},\ a_{2,0},\ a_{3,0},\ a_{0,1},\ a_{1,1},\ \ldots,\ a_{3,3}.$$

These are arranged into a 4×4 matrix

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}.$$

In the following, we'll occasionally need to work with the finite field GF(28). This is covered in Section 3.11. However, for the present purposes, we only need the following facts. The elements of GF(28) are bytes, which consist of eight bits. They can be added

by *XOR*. They can also be multiplied in a certain way (i.e., the product of two bytes is again a byte), but this process is more complicated. Each byte b except the zero byte has a multiplicative inverse; that is, there is a byte b′ such that b·b′=00000001. Since we can do arithmetic operations on bytes, we can work with matrices whose entries are bytes.

As a technical point, we note that the model of GF(28) depends on a choice of irreducible polynomial of degree 8. The choice for Rijndael is X8+X4+X3+X+1. This is also the polynomial used in the examples in Section 3.11. Other choices for this polynomial would presumably give equally good algorithms.

8.2.1 The SubBytes Transformation

In this step, each of the bytes in the matrix is changed to another byte by Table 8.1, called the S-box.

Table 8.1 S-Box for Rijndael

S-Box															
99	124	119	123	242	107	111	197	48	1	103	43	254	215	171	118
202	130	201	125	250	89	71	240	173	212	162	175	156	164	114	192
183	253	147	38	54	63	247	204	52	165	229	241	113	216	49	21
4	199	35	195	24	150	5	154	7	18	128	226	235	39	178	117
9	131	44	26	27	110	90	160	82	59	214	179	41	227	47	132
83	209	0	237	32	252	177	91	106	203	190	57	74	76	88	207
208	239	170	251	67	77	51	133	69	249	2	127	80	60	159	168
81	163	64	143	146	157	56	245	188	182	218	33	16	255	243	210
205	12	19	236	95	151	68	23	196	167	126	61	100	93	25	115
96	129	79	220	34	42	144	136	70	238	184	20	222	94	11	219
224	50	58	10	73	6	36	92	194	211	172	98	145	149	228	121
231	200	55	109	141	213	78	169	108	86	244	234	101	122	174	8
186	120	37	46	28	166	180	198	232	221	116	31	75	189	139	138
112	62	181	102	72	3	246	14	97	53	87	185	134	193	29	158
225	248	152	17	105	217	142	148	155	30	135	233	206	85	40	223
140	161	137	13	191	230	66	104	65	153	45	15	176	84	187	22

Table 8.1 Full Alternative Text

Write a byte as eight bits: *abcdefgh*. Look for the entry in the *abcd* row and *efgh* column (the rows and columns are numbered from 0 to 15). This entry, when converted to binary, is the output.

For example, if the input byte is 10001011, we look in row 8 (the ninth row) and column 11 (the twelfth column). The entry is 61, which is 111101 in binary. This is the output of the S-box.

The output of SubBytes is again a 4×4 matrix of bytes, let's call it

b0, 0b0, 1b0, 2b0, 3b1, 0b1, 1b1, 2b1, 3b2, 0b2, 1b2, 2b2, 3b3, 0b3, 1b3, 2b3,

8.2.2 The ShiftRows Transformation

The four rows of the matrix are shifted cyclically to the left by offsets of 0, 1, 2, and 3, to obtain

c0, 0c0, 1c0, 2c0, 3c1, 0c1, 1c1, 2c1, 3c2, 0c2, 1c2, 2c2, 3c3, 0c3, 1c3, 2c3, 3=

8.2.3 The MixColumns Transformation

Regard a byte as an element of GF(28), as in Section 3.11. Then the output of the ShiftRows step is a 4×4 matrix (ci, j) with entries in GF(28). Multiply this by a matrix, again with entries in GF(28), to produce the output (di, j), as follows:

00000010000000110000000100000001000000010000000110000000
=d0, 0d0, 1d0, 2d0, 3d1, 0d1, 1d1, 2d1, 3d2, 0d2, 1d2, 2d2, 3d3, 0d3, 1d3, 2d3

8.2.4 The RoundKey Addition

The round key, derived from the key in a way we'll describe later, consists of 128 bits, which are arranged in a 4×4 matrix (ki, j) consisting of bytes. This is *XOR*ed with the output of the MixColumns step:

d0, 0d0, 1d0, 2d0, 3d1, 0d1, 1d1, 2d1, 3d2, 0d2, 1d2, 2d2, 3d3, 0d3, 1d3, 2d3,
=e0, 0e0, 1e0, 2e0, 3e1, 0e1, 1e1, 2e1, 3e2, 0e2, 1e2, 2e2, 3e3, 0e3, 1e3, 2e3, 3

This is the final output of the round.

8.2.5 The Key Schedule

The original key consists of 128 bits, which are arranged into a 4×4 matrix of bytes. This matrix is expanded by adjoining 40 more columns, as follows. Label the first four columns W(0), W(1), W(2), W(3). The new columns are generated recursively. Suppose columns up through W(i−1) have been defined. If i is not a multiple of 4, then

$$W(i)=W(i-4)\oplus W(i-1).$$

If i is a multiple of 4, then

$$W(i)=W(i-4)\oplus T(W(i-1)),$$

where T(W(i−1)) is the transformation of W(i−1) obtained as follows. Let the elements of the column W(i−1) be a, b, c, d. Shift these cyclically to obtain b, c, d, a. Now replace each of these bytes with the corresponding element in the S-box from the SubBytes step, to get 4 bytes e, f, g, h. Finally, compute the round constant

$$r(i)=00000010^{(i-4)/4}$$

in GF(28) (recall that we are in the case where i is a multiple of 4). Then T(W(i −1)) is the column vector

$$(e \oplus r(i), f, g, h).$$

In this way, columns W(4), …, W(43) are generated from the initial four columns.

The **round key** for the ith round consists of the columns

W(4i), W(4i+1), W(4i+2), W(4i+3).

8.2.6 The Construction of the S-Box

Although the S-box is implemented as a lookup table, it has a simple mathematical description. Start with a byte x7x6x5x4x3x2x1x0, where each xi is a binary bit. Compute its inverse in GF(28), as in Section 3.11. If the byte is 00000000, there is no inverse, so we use 00000000 in place of its inverse. The resulting byte y7y6y5y4y3y2y1y0 represents an eight-dimensional column vector, with the rightmost bit y0 in the top position. Multiply by a matrix and

add the column vector
(1, 1, 0, 0, 0, 1, 1, 0) to obtain a vector
(z0, z1, z2, z3, z4, z5, z6, z7) as follows:

10001111110001111110001111110001111110000111110000111110000111

The byte z7z6z5z4z3z2z1z0 is the entry in the S-box.

For example, start with the byte 11001011. Its inverse in GF(28) is 00000100, as we calculated in Section 3.11. We now calculate

10001111110001111110001111110001111110000111110000111110000111

This yields the byte 00011111. The first four bits 1100 represent 12 in binary and the last four bits 1011 represent 11 in binary. Add 1 to each of these numbers (since the first row and column are numbered 0) and look in the 13th row

and 12th column of the S-box. The entry is 31, which in binary is 00011111.

Some of the considerations in the design of the S-box were the following. The map $x \mapsto x-1$ was used to achieve nonlinearity. However, the simplicity of this map could possibly allow certain attacks, so it was combined with multiplication by the matrix and adding the vector, as described previously. The matrix was chosen mostly because of its simple form (note how the rows are shifts of each other). The vector was chosen so that no input ever equals its S-box output or the complement of its S-box output (complementation means changing each 1 to 0 and each 0 to 1).

8.3 Decryption

Each of the steps SubBytes, ShiftRows, MixColumns, and AddRoundKey is invertible:

1. The inverse of SubBytes is another lookup table, called **InvSubBytes**.

2. The inverse of ShiftRows is obtained by shifting the rows to the right instead of to the left, yielding **InvShiftRows**.

3. The inverse of MixColumns exists because the 4×4 matrix used in MixColumns is invertible. The transformation **InvMixColumns** is given by multiplication by the matrix

   00001110000010110000110100001001000010010000111000

4. AddRoundKey is its own inverse.

The Rijndael encryption consists of the

steps

ARK

SB, SR, MC, ARK

...

SB, SR, MC, ARK

SB, SR, ARK.

Recall that MC is missing in the last round.

To decrypt, we need to run through the inverses of these steps in the reverse order. This yields the following preliminary version of decryption:

ARK, ISR, ISB

ARK, IMC, ISR, ISB

...

ARK, IMC, ISR, ISB

ARK.

However, we want to rewrite this decryption in order to make it look more like encryption.

Observe that applying SB then SR is the same as first applying SR then SB. This happens because SB acts one byte at a time and SR permutes the bytes. Correspondingly, the order of ISR and ISB can be reversed.

We also want to reverse the order of

ARK and IMC, but this is not possible. Instead, we proceed as follows. Applying MC and then ARK to a matrix (ci, j) is given as

$$(ci, j) \rightarrow (mi, j)(ci, j) \rightarrow (ei, j) = (mi, j)(ci, j) \oplus (ki, j),$$

where (mi, j) is a the 4×4 matrix in MixColumns and (ki, j) is the round key matrix. The inverse is obtained by solving $(ei, j) = (mi, j)(ci, j) \oplus (ki, j)$ for (ci, j) in terms of (ei, j), namely, $(ci, j) = (mi, j)-1(ei, j) \oplus (mi, j)-1(ki, j)$. Therefore, the process is

$$(ei, j) \rightarrow (mi, j)-1(ei, j) \rightarrow (mi, j)-1(ei, j) \oplus (ki, j'),$$

where $(ki, j') = (mi, j)-1(ki, j)$. The first arrow is simply InvMixColumns applied to (ei, j). If we let **InvAddRoundKey** be *XOR*ing with (ki, j′), then we have that the inverse of “MC then ARK” is “

IMC then IARK.” Therefore, we can replace the steps “ARK then IMC” with the steps “IMC then IARK” in the preceding decryption sequence.

We now see that decryption is given by

ARK, ISB, ISR

IMC, IARK, ISB, ISR

...

IMC, IARK, ISB, ISR

ARK.

Regroup the lines to obtain the final version:

Rijndael Decryption
1. ARK, using the 10th round key 2. Nine rounds of ISB, ISR, IMC, IARK, using round keys 9 to 1 3. A final round: ISB, ISR, ARK, using the 0th round key

8.3-3 Full Alternative Text

Therefore, the decryption is given by essentially the same structure as encryption, but SubBytes, ShiftRows, and MixColumns are replaced by their inverses, and AddRoundKey is replaced by InvAddRoundKey, except in the initial and final steps. Of course, the round keys are used in the reverse order, so the first ARK uses the 10th round key, and the last ARK uses the 0th round key.

The preceding shows why the MixColumns is omitted in the last round. Suppose it had been left in. Then the encryption would start ARK, SB,

SR, MC, ARK, ..., and it would end with ARK, SB, SR, MC, ARK. Therefore, the beginning of the decryption would be (after the reorderings) IMC, IARK, ISB, ISR, ... This means the decryption would have an unnecessary IMC at the beginning, and this would have the effect of slowing down the algorithm.

Another way to look at encryption is that there is an initial ARK, then a sequence of alternating half rounds

(SB, SR), (MC, ARK), (SB, SR), …, (MC, ARK), (SB, SR),

followed by a final ARK. The decryption is ARK, followed by a sequence of alternating half rounds

(ISB, ISR), (IMC, IARK), (ISB, ISR), …, (IMC, IARK), (ISB, ISR),

followed by a final ARK. From this point of view, we see that a final MC would not fit naturally into any of the half rounds, and it is natural to leave it out.

On eight-bit processors, decryption is not quite as fast as encryption. This is because the entries in the 4×4 matrix for InvMixColumns are more complex than those for MixColumns, and this is enough to make decryption take around 30% longer than encryption for these processors. However, in many applications, decryption is not needed, for example, when CFB mode (see Section 6.3) is used. Therefore, this is not considered to be a significant drawback.

The fact that encryption and decryption are not identical processes leads to the expectation that there are no weak keys, in contrast to DES (see Exercise 5 in Chapter 7) and several other algorithms.

8.4 Design Considerations

The Rijndael algorithm is not a Feistel system (see Sections 7.1 and 7.2). In a Feistel system, half the bits are moved but not changed during each round. In Rijndael, all bits are treated uniformly. This has the effect of diffusing the input bits faster. It can be shown that two rounds are sufficient to obtain full diffusion, namely, each of the 128 output bits depends on each of the 128 input bits.

The S-box was constructed in an explicit and simple algebraic way so as to avoid any suspicions of trapdoors built into the algorithm. The desire was to avoid the

mysteries about the S-boxes that haunted DES. The Rijndael S-box is highly nonlinear, since it is based on the mapping $x \mapsto x^{-1}$ in GF(28). It is excellent at resisting differential and linear cryptanalysis, as well as more recently studied methods called interpolation attacks.

The ShiftRows step was added to resist two recently developed attacks, namely truncated differentials and the Square attack (Square was a predecessor of Rijndael).

The MixColumns causes diffusion among the bytes. A change in one input byte in this step always results in all four output bytes changing. If two input bytes are changed, at least three output

bytes are changed.

The Key Schedule involves nonlinear mixing of the key bits, since it uses the S-box. The mixing is designed to resist attacks where the cryptanalyst knows part of the key and tries to deduce the remaining bits. Also, it aims to ensure that two distinct keys do not have a large number of round keys in common. The round constants are used to eliminate symmetries in the encryption process by making each round different.

The number of rounds was chosen to be 10 because there are attacks that are better than brute force up to six rounds. No known attack beats brute force for seven or more rounds. It was felt that four extra rounds provide a large enough

margin of safety. Of course, the number of rounds could easily be increased if needed.

8.5 Exercises

1. Suppose the key for round 0 in AES consists of 128 bits, each of which is 0.

    1. Show that the key for the first round is W(4), W(5), W(6), W(7), where

       W(4)=W(5)=W(6)=W(7)=0110001001100011011000110

    2. Show that W(8)=W(10) ≠ W(9)=W(11) (Hint: This can be done without computing W(8) explicitly).

2. Suppose the key for round 0 in AES consists of 128 bits, each of which is 1.

    1. Show that the key for the first round is W(4), W(5), W(6), W(7), where

       W(5)=W(7)=(00010111000101100001011000010110), W

       Note that $W(5)=\overline{W(4)}$ = the complement of W(5) (the complement can be obtained by XORing with a string of all 1s).

2. Show that $W(10)=W(8)^{-}$ and that $W(11)=W(9)^{-}$ (Hints: $W(5) \oplus W(6)$ is a string of all 1s. Also, the relation $A^{-} \oplus B = A \oplus B^{-}$ might be useful.)

3. Let f(x) be a function from binary strings (of a fixed length N) to binary strings. For the purposes of this problem, let's say that f(x) has the *equal difference property* if the following is satisfied: Whenever x1, x2, x3, x4 are binary strings of length N that satisfy $x1 \oplus x2 = x3 \oplus x4$, then

   $f(x1) \oplus f(x2) = f(x3) \oplus f(x4)$.

   1. Show that if $\alpha, \beta \in GF(28)$ and $f(x)=\alpha x + \beta$ for all $x \in GF(28)$, then f(x) has the equal difference property.

   2. Show that the ShiftRows Transformation, the MixColumns Transformation, and the RoundKey Addition have the equal difference property.

4.

   1. Suppose we remove all SubBytes Transformation steps from the AES algorithm. Show that the resulting AES encryption would then have the equal

difference property defined in Exercise 3.

2. Suppose we are in the situation of part (a), with all SubBytes Transformation steps removed. Let x1 and x2 be two 128-bit plaintext blocks and let E(x1) and E(x2) be their encryptions under this modified AES scheme. Show that $E(x1) \oplus E(x2)$ equals the result of encrypting $x1 \oplus x2$ using only the ShiftRows and MixColumns Transformations (that is, both the RoundKey Addition and the SubBytes Transformation are missing). In particular, $E(x1) \oplus E(x2)$ is independent of the key.

3. Suppose we are in the situation of part (a), and Eve knows x1 and E(x1) for some 128-bit string x. Describe how she can decrypt any message E(x2) (your solution should be much faster than using brute force or making a list of all encryptions). (Remark: This shows that the SubBytes transformation is needed to prevent the equal difference property. See also Exercise 5.)

5. Let x1=00000000, x2=00000001, x3=00000010,

x4=00000011. Let SB(x) denote the SubBytes Transformation of x. Show that

$$SB(x1) \oplus SB(x2) = 00011111 \neq 00001100 = SB(x3) \oplus SB(x4).$$

Conclude that the SubBytes Transformation is not an affine map (that is, a map of the form $\alpha x + \beta$) from GF(28) to GF(28). (Hint: See Exercise 3(a).)

6. Your friend builds a very powerful computer that uses brute force to find a 56-bit DES key in 1 hour, so you make an even better machine that can try 256 AES keys in 1 second. How long will this machine take to try all 2128 AES keys?

Chapter 9 The RSA Algorithm

9.1 The RSA Algorithm

Alice wants to send a message to Bob, but they have not had previous contact and they do not want to take the time to send a courier with a key. Therefore, all information that Alice sends to Bob will potentially be obtained by the evil observer Eve. However, it is still possible for a message to be sent in such a way that Bob can read it but Eve cannot.

With all the previously discussed methods, this would be impossible.

Alice would have to send a key, which Eve would intercept. She could then decrypt all subsequent messages. The possibility of the present scheme, called a **public key cryptosystem**, was first publicly suggested by Diffie and Hellman in their classic 1976 paper [Diffie-Hellman]. However, they did not yet have a practical implementation (although they did present an alternative key exchange procedure that works over public channels; see Section 10.4). In the next few years, several methods were proposed. The most successful, based on the idea that factorization of integers into their prime factors is hard, was proposed by Rivest, Shamir, and Adleman in 1977 and is known as the RSA algorithm.

It had long been claimed that government cryptographic agencies had discovered the RSA algorithm several years earlier, but secrecy rules prevented them from releasing any evidence. Finally, in 1997, documents released by CESG, a British cryptographic agency, showed that in 1970, James Ellis had discovered public key cryptography, and in 1973, Clifford Cocks had written an internal document describing a version of the RSA algorithm in which the encryption exponent e (see the discussion that follows) was the same as the modulus n.

Here is how the RSA algorithm works. Bob chooses two distinct large primes p and q and multiplies them together to

form

$$n=pq.$$

He also chooses an encryption exponent e such that

$$\gcd(e,\ (p-1)(q-1))=1.$$

He sends the pair (n, e) to Alice but keeps the values of p and q secret. In particular, Alice, who could possibly be an enemy of Bob, never needs to know p and q to send her message to Bob securely. Alice writes her message as a number m. If m is larger than n, she breaks the message into blocks, each of which is less than n. However, for simplicity, let's assume for the moment that $m < n$. Alice computes

$$c \equiv m^e \pmod{n}$$

and sends c to Bob. Since Bob knows p and q, he can compute (p−1)(q−1) and therefore can find the decryption exponent d with

de ≡ 1 (mod (p−1)(q−1)).

As we'll see later,

m ≡ cd (mod n),

so Bob can read the message.

We summarize the algorithm in the following table.

The RSA Algorithm
1. Bob chooses secret primes p and q and computes $n = pq$. 2. Bob chooses e with $\gcd(e, (p-1)(q-1)) = 1$. 3. Bob computes d with $de \equiv 1 \pmod{(p-1)(q-1)}$. 4. Bob makes n and e public, and keeps p, q, d secret. 5. Alice encrypts m as $c \equiv m^e \pmod{n}$ and sends c to Bob. 6. Bob decrypts by computing $m \equiv c^d \pmod{n}$.

9.1-1 Full Alternative Text

Example

Bob chooses

$p=885320963$, $q=238855417$.

Then

$n=p\cdot q=211463707796206571$.

Let the encryption exponent be

$e=9007$.

The values of n and e are sent to Alice.

Alice's message is *cat*. We will depart from our earlier practice of numbering the letters starting with a=0; instead, we start the numbering at a=01 and continue through z=26. We do this because, in the previous method, if the letter a appeared at the beginning of a message, it would yield a message number m starting with 00, so the a

would disappear.

The message is therefore

$$m=30120.$$

Alice computes

$$c \equiv me \equiv 301209007 \equiv 113535859035722866 (\mathrm{mod}\, n).$$

She sends c to Bob.

Since Bob knows p and q, he knows $(p-1)(q-1)$. He uses the extended Euclidean algorithm (see Section 3.2) to compute d such that

$$de \equiv 1 (\mathrm{mod}\ (p-1)(q-1)).$$

The answer is

$$d=116402471153538991.$$

Bob computes

$$cd \equiv 113535859035722866116402471153538991 \equiv 30120 (\text{mod } n),$$

so he obtains the original message.

For more examples, see Examples 24–30 in the Computer Appendices.

There are several aspects that need to be explained, but perhaps the most important is why $m \equiv cd \pmod{n}$. Recall Euler's theorem (Section 3.6): If $\gcd(a, n)=1$, then $a^{\phi(n)} \equiv 1 \pmod{n}$. In our case, $\phi(n)=\phi(pq)=(p-1)(q-1)$. Suppose $\gcd(m, n)=1$. This is very likely the case; since p and q are large, m probably has neither as a factor. Since $de \equiv 1 \pmod{\phi(n)}$, we can write $de=1+k\phi(n)$ for some integer k. Therefore,

$$cd \equiv (me)d \equiv m^{1+k\phi(n)} \equiv m \cdot (m^{\phi(n)})^k \equiv m \cdot 1^k \equiv m (\text{mod } n).$$

We have shown that Bob can recover the message. If gcd(m, n) ≠ 1, Bob still recovers the message. See Exercise 37.

What does Eve do? She intercepts n, e, c. She does not know p, q, d. We assume that Eve has no way of factoring n. The obvious way of computing d requires knowing $\phi(n)$. We show later that this is equivalent to knowing p and q. Is there another way? We will show that if Eve can find d, then she can probably factor n. Therefore, it is unlikely that Eve finds the decryption exponent d.

Since Eve knows $c \equiv m^e \pmod{n}$, why doesn't she simply take the eth root of c? This works well if we are not working mod n but is very difficult in

our case. For example, if you know that $m^3 \equiv 3 \pmod{85}$, you cannot calculate the cube root of 3, namely 1.4422… , on your calculator and then reduce mod 85. Of course, a case-by-case search would eventually yield m=7, but this method is not feasible for large n.

How does Bob choose p and q? They should be chosen at random, independently of each other. How large depends on the level of security needed, but it seems that they should have at least 300 digits. For reasons that we discuss later, it is perhaps best if they are of slightly different lengths. When we discuss primality testing, we'll see that finding such primes can be done fairly quickly (see also Section 3.6). A

few other tests should be done on p and q to make sure they are not bad. For example, if $p-1$ has only small prime factors, then n is easy to factor by the $p-1$ method (see Section 9.3), so p should be rejected and replaced with another prime.

Why does Bob require $\gcd(e, (p-1)(q-1))=1$? Recall (see Section 3.3) that $de \equiv 1 \pmod{(p-1)(q-1)}$ has a solution d if and only if $\gcd(e, (p-1)(q-1))=1$. Therefore, this condition is needed in order for d to exist. The extended Euclidean algorithm can be used to compute d quickly. Since $p-1$ is even, $e=2$ cannot be used; one might be tempted to use $e=3$. However, there are dangers in using small values of e (see

Section 9.2, Computer Problem 14, and Section 23.3), so something larger is usually recommended. Also, if e is a moderately large prime, then there is no difficulty ensuring that gcd(e, (p−1)(q −1))=1. It is now generally recommended that e ≥ 65537. Among the data collected for [Lenstra2012 et al.] is the distribution of RSA encryption exponents that is given in Table 9.1.

Table 9.1 RSA Encryption Exponents

e	Percentage
65537	95.4933%
17	3.1035%
41	0.4574%
3	0.3578%
19	0.1506%
25	0.1339%
5	0.1111%
7	0.0596%
11	0.0313%
257	0.0241%
other	0.0774%

Table 9.1 Full Alternative Text

In the encryption process, Alice calculates $m^e \pmod n$. Recall that this can be done fairly quickly and without large memory, for example, by successive squaring. See Section 3.5.

This is definitely an advantage of modular arithmetic: If Alice tried to calculate me first, then reduce mod n, it is possible that recording me would overflow her computer's memory. Similarly, the decryption process of calculating $c^d \pmod{n}$ can be done efficiently. Therefore, all the operations needed for encryption and decryption can be done quickly (i.e., in time a power of $\log n$). The security is provided by the assumption that n cannot be factored.

We made two claims. We justify them here. Recall that the point of these two claims was that finding $\phi(n)$ or finding the decryption exponent d is essentially as hard as factoring n. Therefore, if

factoring is hard, then there should be no fast, clever way of finding d.

Claim 1: Suppose n=pq is the product of two distinct primes. If we know n and $\phi(n)$, then we can quickly find p and q.

Note that

$$n-\phi(n)+1=pq-(p-1)(q-1)+1=p+q.$$

Therefore, we know pq and p+q. The roots of the polynomial

$$X2-(n-\phi(n)+1)X+n=X2-(p+q)X+pq=(X-p)(X-q)$$

are p and q, but they can also be calculated by the quadratic formula:

$$p,\ q=(n-\phi(n)+1)\pm(n-\phi(n)+1)2-4n2.$$

This yields p and q.

For example, suppose n=221 and we know that $\phi(n)=192$. Consider the quadratic equation

$$X2-30X+221.$$

The roots are

$$p, q=30\pm302-4\cdot2212=13, 17.$$

For another example, see Example 31 in the Computer Appendices.

Claim 2: If we know d and e, then we can probably factor n.

In the discussion of factorization methods in Section 9.4, we show that if we have an exponent $r > 0$ such that $ar \equiv 1 \pmod{n}$ for several a with $\gcd(a, n)=1$, then we can probably factor n. Since $de-1$ is a multiple of $\phi(n)$, say

$de-1=k\phi(n)$, we have

$$a^{de-1} \equiv (a^{\phi(n)})^k \equiv 1 \pmod{n}$$

whenever gcd(a, n)=1. The $a^r \equiv 1$ factorization method can now be applied.

For an example, see Example 32 in the Computer Appendices.

Claim 2': If e is small or medium-sized (for example, if e has several fewer digits than n) and we know d, then we can factor n.

We use the following procedure:

1. Compute $k=\lceil (de-1)/n \rceil$ (that is, round up to the nearest integer).

2. Compute

$\phi(n) = de - 1k$.

3. Solve the quadratic equation

   $X^2 - (n + 1 - \phi(n))X + n = 0$.

   The solutions are p and q.

Example

Let $n = 670726081$ and $e = 257$. We discover that $d = 524523509$. Compute

$de - 1n = 200.98\cdots$.

Therefore, $k = 201$. Then

$\phi(n) = de - 1201 = 670659412$.

The roots of the equation

$X^2 - (n + 1 - \phi(n))X + n = X^2 - 66670X + 670726081 = 0$

are $12347.00\cdots$ and $54323.00\cdots$, and we can check that $n = 12347 \times 54323$.

Why does this work? We know that

$de \equiv 1 \pmod{(p-1)(q-1)}$, so we write $de=1+(p-1)(q-1)k$. Since $d<(p-1)(q-1)$, we know that

$$(p-1)(q-1)k<de<(p-1)(q-1)e,$$

so $k<e$. We have

$$k=de-1(p-1)(q-1) > de-1n=(p-1)(q-1)kn=(pq-p-q+1)kn=k-(p+q-1)kn.$$

Usually, both p and q are approximately n. In practice, e and therefore k (which is less than e) are much smaller than n. Therefore, $(p+q-1)k$ is much smaller than n, which means that $(p+q-1)k/n<1$ and it rounds off to 0.

Once we have k, we use $de-1=\phi(n)k$ to solve for $\phi(n)$. As we have already seen, once we know n and $\phi(n)$, we can find p and q by solving the quadratic equation.

One way the RSA algorithm can be used is when there are several banks, for example, that want to be able to send financial data to each other. If there are several thousand banks, then it is impractical for each pair of banks to have a key for secret communication. A better way is the following. Each bank chooses integers n and e as before. These are then published in a public book. Suppose bank A wants to send data to bank B. Then A looks up B's n and e and uses them to send the message. In practice, the RSA algorithm is not quite fast enough for sending massive amounts of data. Therefore, the RSA algorithm is often used to send a key for a faster encryption method such as AES.

PGP (= Pretty Good Privacy) used to be a standard method for encrypting email. When Alice sends an email message to Bob, she first signs the message using a digital signature algorithm such as those discussed in Chapter 13. She then encrypts the message using a block cipher such as triple DES or AES (other choices are IDEA or CAST-128) with a randomly chosen 128-bit key (a new random key is chosen for each transmission). She then encrypts this key using Bob's public RSA key (other public key methods can also be used). When Bob receives the email, he uses his private RSA exponent to decrypt the random key. Then he uses this random key to decrypt the message, and he checks the signature to verify that the

message is from Alice. For more discussion of PGP, see Section 15.6.

9.2 Attacks on RSA

In practice, the RSA algorithm has proven to be effective, as long as it is implemented correctly. We give a few possible implementation mistakes in the Exercises. Here are a few other potential difficulties. For more about attacks on RSA, see [Boneh].

Theorem

Let n=pq have m digits. If we know the first m/4, or the last m/4, digits of p, we can efficiently factor n.

In other words, if p and q have 300 digits, and we know the first 150 digits,

or the last 150 digits, of p, then we can factor n. Therefore, if we choose a random starting point to choose our prime p, the method should be such that a large amount of p is not predictable. For example, suppose we take a random 150-digit number N and test numbers of the form $N \cdot 10150+k$, $k=1, 3, 5, \ldots$, for primality until we find a prime p (which should happen for $k<10000$). An attacker who knows that this method is used will know 147 of the last 150 digits (they will all be 0 except for the last three or four digits). Trying the method of the theorem for the various values of $k<10000$ will eventually lead to the factorization of n.

For details of the preceding result, see

[Coppersmith2]. A related result is the following.

Theorem

Suppose (n, e) is an RSA public key and n has m digits. Let d be the decryption exponent. If we have at least the last m/4 digits of d, we can efficiently find d in time that is linear in elog2e.

This means that the time to find d is bounded as a function linear in elog2e. If e is small, it is therefore quite fast to find d when we know a large part of d. If e is large, perhaps around n, the theorem is no better than a case-by-case search for d. For details, see [Boneh et al.].

9.2.1 Low Exponent Attacks

Low encryption or decryption exponents are tempting because they speed up encryption or decryption. However, there are certain dangers that must be avoided. One pitfall of using e=3 is given in Computer Problem 14. Another difficulty is discussed in Chapter 23 (Lattice Methods). These problems can be avoided by using a somewhat higher exponent. One popular choice is e=65537=216+1. This is prime, so it is likely that it is relatively prime to (p−1)(q−1). Since it is one more than a power of 2, exponentiation to this power can be done quickly: To calculate x65537, square x sixteen times, then multiply the result by x.

The decryption exponent d should of course be chosen large enough that brute force will not find it. However, even more care is needed, as the following result shows. One way to obtain desired properties of d is to choose d first, then find e with $de \equiv 1 \pmod{\phi(n)}$.

Suppose Bob wants to be able to decrypt messages quickly, so he chooses a small value of d. The following theorem of M. Wiener [Wiener] shows that often Eve can then find d easily. In practice, if the inequalities in the hypotheses of the proposition are weakened, then Eve can still use the method to obtain d in many cases. Therefore, it is recommended that d be chosen fairly large.

Theorem

Suppose p, q are primes with $q<p<2q$. Let $n=pq$ and let $1 \le d, e<\phi(n)$ satisfy $de \equiv 1 \pmod{(p-1)(q-1)}$. If $d<13n1/4$, then d can be calculated quickly (that is, in time polynomial in logn).

Proof. Since $q2<pq=n$, we have $q<n$. Therefore, since $p<2q$,

$$n-\phi(n)=pq-(p-1)(q-1)=p+q-1<3q<3n.$$

Write $ed=1+\phi(n)k$ for some integer $k \ge 1$. Since $e<\phi(n)$, we have

$$\phi(n)k<ed<13\phi(n)n1/4,$$

so $k<13n1/4$. Therefore,

$$kn-ed=k(n-\phi(n))-1<k(n-\phi(n))<13n1/4(3n)=n3/4.$$

Also, since $k(n-\phi(n))-1 > 0$, we have $kn-ed > 0$. Dividing by dn yields

0<kd−en<1dn1/4<13d2,

since 3d<n1/4 by assumption.

We now need a result about continued fractions. Recall from Section 3.12 that if x is a positive real number and k and d are positive integers with

kd−x<12d2,

then k/d arises from the continued fraction expansion of x. Therefore, in our case, k/d arises from the continued fraction expansion of e/n. Therefore, Eve does the following:

1. Computes the continued fraction of e/n. After each step, she obtains a fraction A/B.

2. Eve uses k=A and d=B to compute C=(ed−1)/k. (Since ed=1+ϕ(n)k, this value of C is a candidate for ϕ(n).)

3. If C is not an integer, she proceeds to the next step

of the continued fraction.

4. If C is an integer, then she finds the roots r1, r2 of $X^2-(n-C+1)X+n$. (Note that this is possibly the equation $X^2-(n-\phi(n)+1)X+n=(X-p)(X-q)$ from earlier.) If r1 and r2 are integers, then Eve has factored n. If not, then Eve proceeds to the next step of the continued fraction algorithm.

Since the number of steps in the continued fraction expansion of e/n is at most a constant times logn, and since the continued fraction algorithm stops when the fraction e/n is reached, the algorithm terminates quickly. Therefore, Eve finds the factorization of n quickly.

Remarks

Recall that the rational approximations to a number x arising from the continued fraction algorithm are alternately larger than x and smaller

than x. Since 0<kd−en, we only need to consider every second fraction arising from the continued fraction.

What happens if Eve reaches e/n without finding the factorization of n? This means that the hypotheses of the proposition are not satisfied. However, it is possible that sometimes the method will yield the factorization of n even when the hypotheses fail.

Example

Let n=1966981193543797 and e=323815174542919. The continued fraction of e/n is

[0;6, 13, 2, 3, 1, 3, 1, 9, 1, 36, 5, 2, 1, 6, 1, 43, 13, 1, 10, 11, 2, 1, 9, 5]=16+113
⋯ .

The first fraction is 1/6, so we try

k=1, d=6. Since d must be odd, we discard this possibility.

By the remark, we may jump to the third fraction:

16+113+12=27164.

Again, we discard this since d must be odd.

The fifth fraction is 121/735. This gives C=(e·735−1)/121, which is not an integer.

The seventh fraction is 578/3511 This gives C=1966981103495136 as the candidate for ϕ(n). The roots of

X2−(n−C+1)X+n

are 37264873 and 52783789, to several

decimal places of accuracy. Since

$$n=37264873\times52783789,$$

we have factored n.

9.2.2 Short Plaintext

A common use of RSA is to transmit keys for use in DES, AES, or other symmetric cryptosystems. However, a naive implementation could lead to a loss of security. Suppose a 56-bit DES key is written as a number $m \approx 1017$. This is encrypted with RSA to obtain $c \equiv m^e \pmod{n}$. Although m is small, the ciphertext c is probably a number of the same size as n, so perhaps around 200 digits. However, Eve attacks the system as follows. She makes two lists:

1. $cx - e \pmod{n}$ for all x with $1 \le x \le 109$.

2. $ye \pmod{n}$ for all y with $1 \le y \le 109$.

She looks for a match between an element on the first list and an element on the second list. If she finds one, then she has $cx - e \equiv ye$ for some x, y. This yields

$$c \equiv (xy)e \pmod{n},$$

so $m \equiv xy \pmod{n}$. Is this attack likely to succeed? Suppose m is the product of two integers x and y, both less than 109. Then these x, y will yield a match for Eve. Not every m will have this property, but many values of m are the product of two integers less than 109. For these, Eve will obtain m.

This attack is much more efficient than

trying all 1017 possibilities for m, which is nearly impossible on one computer, and would take a long time even with several computers working in parallel. In the present attack, Eve needs to compute and store a list of length 109, then compute the elements on the other list and check each one against the first list. Therefore, Eve performs approximately 2×109 computations (and compares with the list up to 109 times). This is easily possible on a single computer. For more on this attack, see [Boneh-Joux-Nguyen].

It is easy to prevent this attack. Instead of using a small value of m, adjoin some random digits to the beginning and end of m so as to form a much longer

plaintext. When Bob decrypts the ciphertext, he simply removes these random digits and obtains m.

A more sophisticated method of preprocessing the plaintext, namely Optimal Asymmetric Encryption Padding (OAEP), was introduced by Bellare and Rogaway [Bellare-Rogaway2] in 1994. Suppose Alice wants to send a message m to Bob, whose RSA public key is (n, e), where n has k bits. Two positive integers k0 and k1 are specified in advance, with $k0+k1<k$. Alice's message is allowed to have $k-k0-k1$ bits. Typical values are $k=1024$, $k0=k1=128$, $k-k0-k1=768$. Let G be a function that inputs strings of k0 bits and outputs strings of $k-k0$ bits. Let

H be a function that inputs k−k0 bits and outputs k0 bits. The functions G and H are usually constructed from hash functions (see Chapter 11 for a discussion of hash functions). To encrypt m, Alice first expands it to length k−k0 by adjoining k1 zero bits. The result is denoted m0k1. She then chooses a random string r of k0 bits and computes

$$x1=m0k1 \oplus G(r),\ x2=r \oplus H(x1).$$

If the concatenation $x1||x2$ is a binary number larger than n, Alice chooses a new random number r and computes new values for x1 and x2. As soon as she obtains $x1||x2<n$ (this has a probability of at least 1/2 of happening for each r, as long as G(r) produces

fairly random outputs), she forms the ciphertext

$$E(m)=(x1\|x2)e(\text{mod } n).$$

To decrypt a ciphertext c, Bob uses his private RSA decryption exponent d to compute $cd \pmod n$. The result is written in the form

$$cd \pmod n=y1\|y2,$$

where y1 has $k-k0$ bits and y2 has k0 bits. Bob then computes

$$m0k1=y1\oplus G(H(y1)\oplus y2).$$

The correctness of this decryption can be justified as follows. If the ciphertext is the encryption of m, then

$$y1=x1=m0k1\oplus G(r)andy2=x2=r\oplus H(x1).$$

Therefore,

$$H(y1) \oplus y2 = H(x1) \oplus r \oplus H(x1) = r$$

and

$$y1 \oplus G(H(y1) \oplus y2) = x1 \oplus G(r) = m0k1.$$

Bob removes the k1 zero bits from the end of m0k1 and obtains m. Also, Bob has check on the integrity of the ciphertext. If there are not k1 zeros at the end, then the ciphertext does not correspond to a valid encryption.

This method is sometimes called plaintext-aware encryption. Note that the padding with x2 depends on the message m and on the random parameter r. This makes chosen ciphertext attacks on the system more difficult. It also is used for ciphertext indistinguishability. See Section 4.5.

For discussion of the security of OAEP, see [Shoup].

9.2.3 Timing Attacks

Another type of attack on RSA and similar systems was discovered by Paul Kocher in 1995, while he was an undergraduate at Stanford. He showed that it is possible to discover the decryption exponent by carefully timing the computation times for a series of decryptions. Though there are ways to thwart the attack, this development was unsettling. There had been a general feeling of security since the mathematics was well understood. Kocher's attack demonstrated that a system could still have unexpected weaknesses.

Here is how the timing attack works. Suppose Eve is able to observe Bob decrypt several ciphertexts y. She times how long this takes for each y. Knowing each y and the time required for it to be decrypted will allow her to find the decryption exponent d. But first, how could Eve obtain such information? There are several situations where encrypted messages are sent to Bob and his computer automatically decrypts and responds. Measuring the response times suffices for the present purposes.

We need to assume that we know the hardware being used to calculate $y^d \pmod{n}$. We can use this information to calculate the computation times for various steps that potentially occur in

the process.

Let's assume that $y^d \pmod n$ is computed by an algorithm given in Exercise 56 in Chapter 3, which is as follows:

Let $d=b_1b_2\ldots b_w$ be written in binary (for example, when $x=1011$, we have $b_1=1$, $b_2=0$, $b_3=1$, $b_4=1$). Let y and n be integers. Perform the following procedure:

1. Start with $k=1$ and $s_1=1$.
2. If $b_k=1$, let $r_k \equiv s_k y \pmod n$. If $b_k=0$, let $r_k=s_k$.
3. Let $s_{k+1} \equiv r_k^2 \pmod n$.
4. If $k=w$, stop. If $k<w$, add 1 to k and go to (2).

Then $r_w \equiv y^d \pmod n$.

Note that the multiplication sky occurs only when the bit bk=1. In many situations, there is a reasonably large variation in how long this multiplication takes. We assume this is the case here.

Before we continue, we need a few facts from probability. Suppose we have a random process that produces real numbers t as outputs. For us, t will be the time it takes for the computer to complete a calculation, given a random input y. The mean is the average value of these outputs. If we record outputs t1, …, tn, the mean should be approximately m=(t1+⋯tn)/n. The variance for the random process is approximated by

Var({ti})=(t1−m)2+⋯+(tn−m)2n.

The standard deviation is the square root of the variance and gives a measure of how much variation there is in the values of the ti's.

The important fact we need is that when two random processes are independent, the variance for the sum of their outputs is the sum of the variances of the two processes. For example, we will break the computation done by the computer into two independent processes, which will take times t' and t''. The total time t will be $t'+t''$. Therefore, Var({ti}) should be approximately Var({ti′})+Var({ti″}).

Now assume Eve knows ciphertexts $y1, \ldots, yn$ and the times that it took to compute each $yid \pmod n$. Suppose she knows bits $b1, \ldots, bk-1$ of the exponent

d. Since she knows the hardware being used, she knows how much time was used in calculating r1, …, rk−1 in the preceding algorithm. Therefore, she knows, for each yi, the time ti that it takes to compute rk, …, rw.

Eve wants to determine bk. If bk=1, a multiplication sky (mod n) will take place for each ciphertext yi that is processed. If bk=0, there is no such multiplication.

Let ti′ be the amount of time it takes the computer to perform the multiplication sky (mod n), though Eve does not yet know whether this multiplication actually occurs. Let ti″=ti−ti′. Eve computes Var({ti}) and Var({ti″}). If Var({ti}) > Var({ti″}), then Eve

concludes that bk=1. If not, bk=0. After determining bk, she proceeds in the same manner to find all the bits.

Why does this work? If the multiplication occurs, ti″ is the amount of time it takes the computer to complete the calculation after the multiplication. It is reasonable to assume ti′ and ti″ are outputs that are independent of each other. Therefore,

$$\mathrm{Var}(\{ti\}) \approx \mathrm{Var}(\{ti'\})+\mathrm{Var}(\{ti''\}) > \mathrm{Var}(\{ti''\}).$$

If the multiplication does not occur, ti′ is the amount of time for an operation unrelated to the computation, so it is reasonable to assume ti and ti′ are independent. Therefore,

$$\mathrm{Var}(\{ti''\}) \approx \mathrm{Var}(\{ti\})+\mathrm{Var}(\{-ti'\}) > \mathrm{Var}(\{ti\}).$$

Note that we couldn't use the mean in place of the variance, since the mean of $\{-t_i\}$ would be negative, so the last inequality would not hold. All that can be deduced from the mean is the total number of nonzero bits in the binary expansion of d.

The preceding gives a fairly simple version of the method. In practice, various modifications would be needed, depending on the specific situation. But the general strategy remains the same. For more details, see [Kocher]. For more on timing attacks, see [Crosby et al.].

A similar attack on RSA works by measuring the power consumed during the computations. See [Kocher et al.].

Another method, called acoustic cryptanalysis, obtains information from the high-pitched noises emitted by the electronic components of a computer during its computations. See [Genkin et al.]. Attacks such as these and the timing attack can be prevented by appropriate design features in the physical implementation.

Timing attacks, power analysis, and acoustic cryptanalysis are examples of **side-channel attacks**, where the attack is on the implementation rather than on the basic cryptographic algorithm.

9.3 Primality Testing

Suppose we have an integer of 300 digits that we want to test for primality. We know by Exercise 7 in Chapter 3 that one way is to divide by all the primes up to its square root. What happens if we try this? There are around 3×10^{147} primes less than 10^{150}. This is significantly more than the number of particles in the universe. Moreover, if the computer can handle 1010 primes per second, the calculation would take around 3×10^{130} years. (It's been suggested that you could go sit on the beach for 20 years, then buy a computer that is 1000 times faster, which would cut the runtime down to 3×10^{127} years

– a very large savings!) Clearly, better methods are needed. Some of these are discussed in this section.

A very basic idea, one that is behind many factorization methods, is the following.

Basic Principle

Let n be an integer and suppose there exist integers x and y with $x^2 \equiv y^2 \pmod{n}$, but $x \not\equiv \pm y \pmod{n}$. Then n is composite. Moreover, gcd(x−y, n) gives a nontrivial factor of n.

Proof. Let d=gcd(x−y, n). If d=n then $x \equiv y \pmod{n}$, which is assumed not to happen. Suppose d=1. The Proposition in Subsection 3.3.1 says that if $ab \equiv ac$

(mod n) and if gcd(a, n)=1, then b ≡ c (mod n). In our case, let a=x−y, let b=x +y, and let c=0. Then ab=x2−y2 ≡ 0 ≡ ac. If d=gcd(x−y, n)=1, then x+y=b ≡ c=0. This says that x ≡ −y (mod n), which contradicts the assumption that x ≡ −y (mod n). Therefore, d ≠ 1, n, so d is a nontrivial factor of n.

Example

Since 122 ≡ 22 (mod 35), but 12 ≡ ±2 (mod 35), , we know that 35 is composite. Moreover, gcd(12−2, 35)=5 is a nontrivial factor of 35.

It might be surprising, but factorization and primality testing are not the same. It is much easier to prove a number is

composite than it is to factor it. There are many large integers that are known to be composite but that have not been factored. How can this be done? We give a simple example. We know by Fermat's theorem that if p is prime, then $2^{p-1} \equiv 1 \pmod{p}$. Let's use this to show 35 is not prime. By successive squaring, we find (congruences are mod 35)

$$2^4 \equiv 16,\ 2^8 \equiv 256 \equiv 11\ 2^{16} \equiv 121 \equiv 16\ 2^{32} \equiv 256 \equiv 11.$$

Therefore,

$$2^{34} \equiv 2^{32}2^2 \equiv 11 \cdot 4 \equiv 9 \equiv 1 \pmod{35}.$$

Fermat's theorem says that 35 cannot be prime, so we have proved 35 to be composite without finding a factor.

The same reasoning gives us the following.

Fermat Primality Test

Let n > 1 be an integer. Choose a random integer a with 1<a<n−1. If an −1 ≡ 1 (mod n), then n is composite. If an−1 ≡ 1 (mod n), then n is probably prime.

Although this and similar tests are usually called “primality tests,” they are actually “compositeness tests,” since they give a completely certain answer only in the case when n is composite. The Fermat test is quite accurate for large n. If it declares a number to be composite, then this is guaranteed to be true. If it declares a number to be probably prime, then empirical results show that this is very likely true. Moreover, since modular exponentiation

is fast, the Fermat test can be carried out quickly.

Recall that modular exponentiation is accomplished by successive squaring. If we are careful about how we do this successive squaring, the Fermat test can be combined with the Basic Principle to yield the following stronger result.

Miller-Rabin Primality Test

Let $n > 1$ be an odd integer. Write $n - 1 = 2^k m$ with m odd. Choose a random integer a with $1 < a < n-1$. Compute $b_0 \equiv a^m \pmod{n}$. If $b_0 \equiv \pm 1 \pmod{n}$, then stop and declare that n is probably prime. Otherwise, let $b_1 \equiv b_0^2 \pmod{n}$. If $b_1 \equiv 1 \pmod{n}$, then n is composite (and $\gcd(b_0 - 1, n)$ gives a nontrivial

factor of n). If $b1 \equiv -1 \pmod{n}$, then stop and declare that n is probably prime. Otherwise, let $b2 \equiv b12 \pmod{n}$. If $b2 \equiv 1 \pmod{n}$, then n is composite. If $b2 \equiv -1 \pmod{n}$, then stop and declare that n is probably prime. Continue in this way until stopping or reaching $bk-1$. If $bk-1 \equiv -1 \pmod{n}$, then n is composite.

Example

Let $n=561$. Then $n-1=560=16 \cdot 35$, so $2k=24$ and $m=35$. Let $a=2$. Then

$b0 \equiv 235 \equiv 263 \pmod{561} b1 \equiv b02 \equiv 166 \pmod{561} b2 \equiv b12 \equiv 67 \pmod{561} b3 \equiv b22 \equiv 1 \pmod{561}$.

Since $b3 \equiv 1 \pmod{561}$, we conclude that 561 is composite. Moreover, $\gcd(b2-1, 561)=33$, which is a

nontrivial factor of 561.

If n is composite and $a^{n-1} \equiv 1 \pmod{n}$, then we say that n is a pseudoprime for the base a. If a and n are such that n passes the Miller-Rabin test, we say that n is a strong pseudoprime for the base a. We showed in Section 3.6 that $2^{560} \equiv 1 \pmod{561}$, so 561 is a pseudoprime for the base 2. However, the preceding calculation shows that 561 is not a strong pseudoprime for the base 2. For a given base, strong pseudoprimes are much more rare than pseudoprimes.

Up to 10^{10}, there are 455052511 primes. There are 14884 pseudoprimes for the base 2, and 3291 strong pseudoprimes for the base 2. Therefore, calculating $2^{n-1} \pmod{n}$ will fail to

recognize a composite in this range with probability less than 1 out of 30 thousand, and using the Miller-Rabin test with a=2 will fail with probability less than 1 out of 100 thousand.

It can be shown that the probability that the Miller-Rabin test fails to recognize a composite for a randomly chosen a is at most 1/4. In fact, it fails much less frequently than this. See [Damgå rd et al.]. If we repeat the test 10 times, say, with randomly chosen values of a, then we expect that the probability of certifying a composite number as prime is at most (1/4)10≅10−6. In practice, using the test for a single a is fairly accurate.

Though strong pseudoprimes are rare, it

has been proved (see [Alford et al.]) that, for any finite set B of bases, there are infinitely many integers that are strong pseudoprimes for all $b \in B$. The first strong pseudoprime for all the bases b=2, 3, 5, 7 is 3215031751. There is a 337-digit number that is a strong pseudoprime for all bases that are primes <200.

Suppose we need to find a prime of around 300 digits. The prime number theorem asserts that the density of primes around x is approximately 1/lnx. When x=10300, this gives a density of around 1/ln(10300)=1/690. Since we can skip the even numbers, this can be raised to 1/345. Pick a random starting point, and throw out the even numbers

(and multiples of other small primes). Test each remaining number in succession by the Miller-Rabin test. This will tend to eliminate all the composites. On average, it will take less than 400 uses of the Miller-Rabin test to find a likely candidate for a prime, so this can be done fairly quickly. If we need to be completely certain that the number in question is prime, there are more sophisticated primality tests that can test a number of 300 digits in a few seconds.

Why does the test work? Suppose, for example, that $b^3 \equiv 1 \pmod{n}$. This means that $b^{22} \equiv 12 \pmod{n}$. Apply the Basic Principle from before. Either $b^2 \equiv \pm 1 \pmod{n}$, or $b^2 \equiv \pm 1 \pmod{n}$ and n is

composite. In the latter case, gcd(b2−1, n) gives a nontrivial factor of n. In the former case, the algorithm would have stopped by the previous step. If we reach bk−1, we have computed bk−1 ≡ a(n−1)/2 (mod n). The square of this is an−1, which must be 1 (mod n) if n is prime, by Fermat's theorem. Therefore, if n is prime, bk −1 ≡ ±1 (mod n). All other choices mean that n is composite. Moreover, if bk −1 ≡ 1, then, if we didn't stop at an earlier step, bk−22 ≡ 12 (mod n) with bk −2 ≡ ±1 (mod n). This means that n is composite (and we can factor n).

In practice, if n is composite, usually we reach bk−1 and it is not ±1 (mod n). In fact, usually an−1 ≡ 1 (mod n). This

means that Fermat's test says n is not prime.

For example, let n=299 and a=2. Since $2298 \equiv 140 \pmod{299}$, Fermat's theorem and also the Miller-Rabin test say that 299 is not prime (without factoring it). The reason this happens is the following. Note that $299=13\times23$. An easy calculation shows that $212 \equiv 1 \pmod{13}$ and no smaller exponent works. In fact, $2j \equiv 1 \pmod{13}$ if and only if j is a multiple of 12. Since 298 is not a multiple of 12, we have $2298 \equiv 1 \pmod{13}$, and therefore also $2298 \equiv 1 \pmod{299}$. Similarly, $2j \equiv 1 \pmod{23}$ if and only if j is a multiple of 11, from which we can again deduce that $2298 \equiv 1 \pmod{299}$. If Fermat's

theorem (and the Miller-Rabin test) were to give us the wrong answer in this case, we would have needed $13 \cdot 23 - 1$ to be a multiple of $12 \cdot 11$.

Consider the general case $n=pq$, a product of two primes. For simplicity, consider the case where $p > q$ and suppose $ak \equiv 1 \pmod{p}$ if and only if $k \equiv 0 \pmod{p-1}$. This means that a is a primitive root mod p; there are $\phi(p-1)$ such a mod p. Since $0<q-1<p-1$, we have

$$n-1 \equiv pq-1 \equiv q(p-1)+q-1 \equiv 0 \pmod{p-1}.$$

Therefore, $a^{n-1} \equiv 1 \pmod{p}$ by our choice of a, which implies that $a^{n-1} \equiv 1 \pmod{n}$. Similar reasoning shows that usually $a^{n-1} \equiv 1 \pmod{n}$ for many other choices of a, too.

But suppose we are in a case where an $-1 \equiv 1 \pmod{n}$. What happens? Let's look at the example of n=561. Since $561=3\times11\times17$, we consider what is happening to the sequence b0, b1, b2, b3 mod 3, mod 11, and mod 17:

$b0 \equiv -1 \pmod 3, \equiv -1 \pmod{11}, \equiv 8 \pmod{17}$, $b1 \equiv 1 \pmod 3, \equiv 1 \pmod{11}, \equiv -4 \pmod{17}$, $b2 \equiv 1 \pmod 3, \equiv 1 \pmod{11}, \equiv -1 \pmod{17}$, $b3 \equiv 1 \pmod 3, \equiv 1 \pmod{11}, \equiv 1 \pmod{17}$.

Since $b3 \equiv 1 \pmod{561}$, we have $b22 \equiv b3 \equiv 1$ mod all three primes. But there is no reason that b3 is the first time we get $bi \equiv 1$ mod a particular prime. We already have $b1 \equiv 1$ mod 3 and mod 11, but we have to wait for b3 when working mod 17. Therefore, $b22 \equiv b3 \equiv 1$ mod 3, mod 11, and mod 17, but b2 is congruent to 1 only mod 3 and mod 11. Therefore, $b2-1$ contains

the factors 3 and 11, but not 17. This is why gcd(b2−1, 561) finds the factor 33 of 561. The reason we could factor 561 by this method is that the sequence b0, b1, … reached 1 mod the primes not all at the same time.

More generally, consider the case n=pq (a product of several primes is similar) and suppose $a^{n-1} \equiv 1 \pmod{n}$. As pointed out previously, it is very unlikely that this is the case; but if it does happen, look at what is happening mod p and mod q. It is likely that the sequences $b_i \pmod{p}$ and $b_i \pmod{q}$ reach −1 and then 1 at different times, just as in the example of 561. In this case, we will be have $b_i \equiv -1 \pmod{p}$ but $b_i \equiv 1 \pmod{q}$ for some i; therefore,

$b_i^2 \equiv 1 \pmod{n}$ but $b_i \not\equiv \pm 1 \pmod{n}$. Therefore, we'll be able to factor n.

The only way that n can pass the Miller-Rabin test is to have $a^{n-1} \equiv 1 \pmod{n}$ and also to have the sequences $b_i \pmod{p}$ and $b_i \pmod{q}$ reach 1 at the same time. This rarely happens.

Another primality test of a nature similar to the Miller-Rabin test is the following, which uses the Jacobi symbol (see Section 3.10).

Solovay-Strassen Primality Test

Let n be an odd integer. Choose several random integers a with $1<a<n-1$. If

$$\left(\frac{a}{n}\right) \equiv a^{(n-1)/2} \pmod{n}$$

for some a, then n is composite. If

$$(an) \equiv a(n-1)/2 (\mathrm{mod}\ n)$$

for all of these random a, then n is probably prime.

Note that if n is prime, then the test will declare n to be a probable prime. This is because of Part 2 of the second Proposition in Section 3.10.

The Jacobi symbol can be evaluated quickly, as in Section 3.10. The modular exponentiation can also be performed quickly.

For example,

$$(215) = -1 \equiv 23 \equiv 2(15-1)/2 (\mathrm{mod}\ 15),$$

so 15 is not prime. As in the Miller-

Rabin tests, we usually do not get ±1 for $a^{(n-1)/2} \pmod n$. Here is a case where it happens:

$$(2341) = -1 \equiv +1 \equiv 2^{(341-1)/2} \pmod{341}.$$

However, the Solovay-Strassen test says that 341 is composite.

Both the Miller-Rabin and the Solovay-Strassen tests work quickly in practice, but, when p is prime, they do not give rigorous proofs that p is prime. There are tests that actually prove the primality of p, but they are somewhat slower and are used only when it is essential that the number be proved to be prime. Most of these methods are probabilistic, in the sense that they work with very high probability in any given case, but success is not guaranteed. In 2002,

Agrawal, Kayal, and Saxena [Agrawal et al.] gave what is known as a deterministic polynomial time algorithm for deciding whether or not a number is prime. This means that the computation time is always, rather than probably, bounded by a constant times a power of logp. This was a great theoretical advance, but their algorithm has not yet been improved to the point that it competes with the probabilistic algorithms.

For more on primality testing and its history, see [Williams].

9.4 Factoring

We now turn to factoring. The basic method of dividing an integer n by all primes $p \leq n$ is much too slow for most purposes. For many years, people have worked on developing more efficient algorithms. We present some of them here. In Chapter 21, we'll also cover a method using elliptic curves, and in Chapter 25, we'll show how a quantum computer, if built, could factor efficiently.

One method, which is also too slow, is usually called the **Fermat factorization** method. The idea is to express n as a difference of two squares: $n=x2-y2$.

Then $n=(x+y)(x-y)$ gives a factorization of n. For example, suppose we want to factor $n=295927$. Compute $n+12$, $n+22$, $n+32$, …, until we find a square. In this case, $295927+32=295936=544^2$. Therefore,

$$295927=(544+3)(544-3)=547\cdot 541.$$

The Fermat method works well when n is the product of two primes that are very close together. If $n=pq$, it takes $|p-q|/2$ steps to find the factorization. But if p and q are two randomly selected 300-digit primes, it is likely that $|p-q|$ will be very large, probably around 300 digits, too. So Fermat factorization is unlikely to work. Just to be safe, however, the primes for an RSA modulus are often chosen to be of

slightly different sizes.

We now turn to more modern methods. If one of the prime factors of n has a special property, it is sometimes easier to factor n. For example, if p divides n and p−1 has only small prime factors, the following method is effective. It was invented by Pollard in 1974.

The p−1 Factoring Algorithm

Choose an integer a > 1. Often a=2 is used. Choose a bound B. Compute $b \equiv aB! \pmod{n}$ as follows. Let $b1 \equiv a \pmod{n}$ and $bj \equiv bj-1j \pmod{n}$. Then $bB \equiv b \pmod{n}$. Let d=gcd(b−1, n). If 1<d<n, we have found a nontrivial factor of n.

Suppose p is a prime factor of n such that $p-1$ has only small prime factors. Then it is likely that $p-1$ will divide $B!$, say $B!=(p-1)k$. By Fermat's theorem, $b \equiv aB! \equiv (ap-1)k \equiv 1 \pmod{p}$, so p will occur in the greatest common divisor of $b-1$ and n. If q is another prime factor of n, it is unlikely that $b \equiv 1 \pmod{q}$, unless $q-1$ also has only small prime factors. If $d=n$, not all is lost. In this case, we have an exponent r (namely $B!$) and an a such that $a^r \equiv 1 \pmod{n}$. There is a good chance that the $a^r \equiv 1$ method (explained later in this section) will factor n. Alternatively, we could choose a smaller value of B and repeat the calculation.

For an example, see Example 34 in the

How do we choose the bound B? If we choose a small B, then the algorithm will run quickly but will have a very small chance of success. If we choose a very large B, then the algorithm will be very slow. The actual value used will depend on the situation at hand.

In the applications, we will use integers that are products of two primes, say $n=pq$, but that are hard to factor. Therefore, we should ensure that $p-1$ has at least one large prime factor. This is easy to accomplish. Suppose we want p to have around 300 digits. Choose a large prime p_0, perhaps around 10^{140}. Look at integers of the form kp_0+1, with k running through some integers

around 10160. Test kp0+1 for primality by the Miller-Rabin test, as before. On the average, this should produce a desired value of p in less than 400 steps. Now choose a large prime q0 and follow the same procedure to obtain q. Then n=pq will be hard to factor by the p−1 method.

The elliptic curve factorization method (see Section 21.3) gives a generalization of the p−1 method. However, it uses some random numbers near p−1 and only requires at least one of them to have only small prime factors. This allows the method to detect many more primes p, not just those where p−1 has only small prime factors.

9.4.1 x2≡y2

Since it is the basis of the best current factorization methods, we repeat the following result from Section 9.4.

Basic Principle

Let n be an integer and suppose there exist integers x and y with x2 ≡ y2 (mod n), but x ≡ ±y (mod n). Then n is composite. Moreover, gcd(x−y, n) gives a nontrivial factor of n.

For an example, see Example 33 in the Computer Appendices.

How do we find the numbers x and y? Let's suppose we want to factor n=3837523. Observe the following:

$93982 \equiv 55 \cdot 19 \pmod{3837523}$
$190952 \equiv 22 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \pmod{3837523}$
$19642 \equiv 32 \cdot 133 \pmod{3837523}$
$170782 \equiv 26 \cdot 32 \cdot 11 \pmod{3837523}$.

If we multiply the relations, we obtain

$$(9398 \cdot 19095 \cdot 1964 \cdot 17078)^2 \equiv (24 \cdot 32 \cdot 53 \cdot 11 \cdot 132 \cdot 19)^2 \equiv 22303872 \equiv 25867052.$$

Since $2230387 \equiv \pm 2586705 \pmod{3837523}$, we now can factor 3837523 by calculating

$$\gcd(2230387 - 2586705, 3837523) = 1093.$$

The other factor is $3837523/1093 = 3511$.

Here is a way of looking at the calculations we just did. First, we generate squares such that when they are reduced mod $n = 3837523$ they can be written as products of small primes (in the present case, primes less than 20). This set of primes is called our **factor base**. We'll discuss how to generate

such squares shortly. Each of these squares gives a row in a matrix, where the entries are the exponents of the primes 2, 3, 5, 7, 11, 13, 17, 19. For example, the relation $170782 \equiv 26 \cdot 32 \cdot 11 \pmod{3837523}$ gives the row 6, 2, 0, 0, 1, 0, 0, 0.

In addition to the preceding relations, suppose that we have also found the following relations:

$80772 \equiv 2 \cdot 19 \pmod{3837523}$ $33972 \equiv 25 \cdot 5 \cdot 132 \pmod{3837523}$ $142622 \equiv 52 \cdot 72 \cdot 13 \pmod{3837523}$.

We obtain the matrix

939819095196417078807733971426200500001201011010200030062001000

Now look for linear dependencies mod 2 among the rows. Here are three of them:

1. 1st + 5th + 6th = $(6,0,6,0,0,2,0,2) \equiv 0 \pmod{2}$

2. 1st + 2nd + 3rd + 4th = (8,4,6,0,2,4,0,2) ≡ 0 (mod 2)

3. 3rd + 7th = (0,2,2,2,0,4,0,0) ≡ 0 (mod 2)

When we have such a dependency, the product of the numbers yields a square. For example, these three yield

1. (9398·8077·3397)2 ≡ 26·56·132·192 ≡ (23·53·13·19)2

2. (9398·19095·1964·17078)2 ≡ (23·32·53·11·132·19)2

3. (1964·14262)2 ≡ (3·5·7·132)2

Therefore, we have x2 ≡ y2 (mod n) for various values of x and y. If x ≡ ±y (mod n), then gcd(x−y, n) yields a nontrivial factor of n. If x ≡ ±y (mod n), then usually gcd(x−y, n)=1 or n, so we don't obtain a factorization. In our three examples, we have

1. 35905232 ≡ 2470002, but 3590523 ≡ −247000

(mod 3837523)

2. 22303872 ≡ 25867052 and
   gcd(2230387−2586705, 3837523)=1093

3. 11479072 ≡ 177452 and
   gcd(1147907−17745, 3837523)=1093

We now return to the basic question: How do we find the numbers 9398, 19095, etc.? The idea is to produce squares that are slightly larger than a multiple of n, so they are small mod n. This means that there is a good chance they are products of small primes. An easy way is to look at numbers of the form [in+j] for small j and for various values of i. Here [x] denotes the greatest integer less than or equal to x. The square of such a number is approximately in+2jin+j2, which is approximately 2jin+j2 mod n. As long

as i is not too large, this number is fairly small, hence there is a good chance it is a product of small primes.

In the preceding calculation, we have 8077=[17n+1] and 9398=[23n+4], for example.

The method just used is the basis of many of the best current factorization methods. The main step is to produce congruence relations

x2 ≡ product of small primes.

An improved version of the above method is called the quadratic sieve. A recent method, the number field sieve, uses more sophisticated techniques to produce such relations and is somewhat faster in many situations. See

[Pomerance] for a description of these two methods and for a discussion of the history of factorization methods. See also Exercise 52.

Once we have several congruence relations, they are put into a matrix, as before. If we have more rows than columns in the matrix, we are guaranteed to have a linear dependence relation mod 2 among the rows. This leads to a congruence $x2 \equiv y2 \pmod{n}$. Of course, as in the case of 1st + 5th + 6th $\equiv 0 \pmod{2}$ considered previously, we might end up with $x \equiv \pm y$, in which case we don't obtain a factorization. But this situation is expected to occur at most half the time. So if we have enough relations – for example, if there

are several more rows than columns – then we should have a relation that yields $x2 \equiv y2$ with $x \equiv \pm y$. In this case $\gcd(x-y, n)$ is a nontrivial factor of n.

In the last half of the twentieth century, there was dramatic progress in factoring. This was partly due to the development of computers and partly due to improved algorithms. A major impetus was provided by the use of factoring in cryptology, especially the RSA algorithm. Table 9.2 gives the factorization records (in terms of the number of decimal digits) for various years.

Table 9.2 Factorization Records

Year	Number of Digits
1964	20
1974	45
1984	71
1994	129
1999	155
2003	174
2005	200
2009	232

Table 9.2 Full Alternative Text

9.4.2 Using ar≡1

On the surface, the Miller-Rabin test looks like it might factor n quite often; but what usually happens is that bk−1 is reached without ever having bu ≡ ±1 (mod n). The problem is that usually an −1 ≡ 1 (mod n). Suppose, on the other hand, that we have some exponent r, maybe not n−1, such that ar ≡ 1 (mod n)

for some a with gcd(a, n)=1. Then it is often possible to factor n.

$a^r \equiv 1$ Factorization Method

Suppose we have an exponent $r > 0$ and an integer a such that $a^r \equiv 1 \pmod{n}$. Write $r = 2^k m$ with m odd. Let $b_0 \equiv a^m \pmod{n}$, and successively define $b_{u+1} \equiv b_u^2 \pmod{n}$ for $0 \le u \le k-1$. If $b_0 \equiv 1 \pmod{n}$, then stop; the procedure has failed to factor n. If, for some u, we have $b_u \equiv -1 \pmod{n}$, stop; the procedure has failed to factor n. If, for some u, we have $b_{u+1} \equiv 1 \pmod{n}$ but $b_u \not\equiv \pm 1 \pmod{n}$, then $\gcd(b_u - 1, n)$ gives a nontrivial factor of n.

Of course, if we take a=1, then any r works. But then $b_0 = 1$, so the method

fails. But if a and r are found by some reasonably sensible method, there is a good chance that this method will factor n.

This looks very similar to the Miller-Rabin test. The difference is that the existence of r guarantees that we have $bu+1 \equiv 1 \pmod{n}$ for some u, which doesn't happen as often in the Miller-Rabin situation. Trying a few values of a has a very high probability of factoring n.

Of course, we might ask how we can find an exponent r. Generally, this seems to be very difficult, and this test cannot be used in practice. However, it is useful in showing that knowing the decryption exponent in the RSA algorithm allows

us to factor the modulus. Moreover, if a quantum computer is built, it will perform factorizations by finding such an exponent r via its unique quantum properties. See Chapter 25.

For an example for how this method is used in analyzing RSA, see Example 32 in the Computer Appendices.

9.5 The RSA Challenge

When the RSA algorithm was first made public in 1977, Rivest, Shamir, and Adleman made the following challenge.

Let the RSA modulus be

n=1143816257578888676692357799761466120102182967212423625625618

and let e=9007 be the encryption exponent. The ciphertext is

c=9686961375462206147714092225435588290575999112457431987469512

Find the message.

The only known way of finding the plaintext is to factor n. In 1977, it was estimated that the then-current

factorization methods would take 4×1016 years to do this, so the authors felt safe in offering $100 to anyone who could decipher the message before April 1, 1982. However, techniques have improved, and in 1994, Atkins, Graff, Lenstra, and Leyland succeeded in factoring n.

They used 524339 “small” primes, namely those less than 16333610, plus they allowed factorizations to include up to two “large” primes between 16333610 and 230. The idea of allowing large primes is the following: If one large prime q appears in two different relations, these can be multiplied to produce a relation with q squared. Multiplying by $q^{-2} \pmod n$ yields a

relation involving only small primes. In the same way, if there are several relations, each with the same two large primes, a similar process yields a relation with only small primes. The "birthday paradox" (see Section 12.1) implies that there should be several cases where a large prime occurs in more than one relation.

Six hundred people, with a total of 1600 computers working in spare time, found congruence relations of the desired type. These were sent by e-mail to a central machine, which removed repetitions and stored the results in a large matrix. After seven months, they obtained a matrix with 524339 columns and 569466 rows. Fortunately, the matrix was sparse, in

the sense that most of the entries of the matrix were 0s, so it could be stored efficiently. Gaussian elimination reduced the matrix to a nonsparse matrix with 188160 columns and 188614 rows. This took a little less than 12 hours. With another 45 hours of computation, they found 205 dependencies. The first three yielded the trivial factorization of n, but the fourth yielded the factors

p=34905295108476509491478496199038981334177646384933878439908205
q=32769132993266709549961988190834461413177642967992942539798288

Computing $9007^{-1} \pmod{(p-1)(q-1)}$ gave the decryption exponent

d =1066986143685780244428687713289201547807099066339378628012262

Calculating $c^d \pmod{n}$ yielded the plaintext message

2008050013010709030023151804190001180500191721050113091908001519

which, when changed back to letters using a=01, b=02, …, blank=00, yielded

the magic words are squeamish ossifrage

(a squeamish ossifrage is an overly sensitive hawk; the message was chosen so that no one could decrypt the message by guessing the plaintext and showing that it encrypted to the ciphertext). For more details of this factorization, see [Atkins et al.]. If you want to see how the decryption works once the factorization is known, see Example 28 in the Computer Appendices.

9.6 An Application to Treaty Verification

Countries A and B have signed a nuclear test ban treaty. Now each wants to make sure the other doesn't test any bombs. How, for example, is country A going to use seismic data to monitor country B? Country A wants to put sensors in B, which then send data back to A. Two problems arise.

1. Country A wants to be sure that Country B doesn't modify the data.

2. Country B wants to look at the message before it's sent to be sure that nothing else, such as espionage data, is being transmitted.

These seemingly contradictory

requirements can be met by reversing RSA. First, A chooses $n=pq$ to be the product of two large primes and chooses encryption and decryption exponents e and d. The numbers n and e are given to B, but p, q, and d are kept secret. The sensor (it's buried deep in the ground and is assumed to be tamper proof) collects the data x and uses d to encrypt x to $y \equiv x^d \pmod{n}$. Both x and y are sent first to country B, which checks that $y^e \equiv x \pmod{n}$. If so, it knows that the encrypted message y corresponds to the data x, and forwards the pair x, y to A. Country A then checks that $y^e \equiv x \pmod{n}$, also. If so, A can be sure that the number x has not been modified, since if x is chosen, then solving $y^e \equiv x \pmod{n}$ for y is the same as decrypting

the RSA message x, and this is believed to be hard to do. Of course, B could choose a number y first, then let $x \equiv y^e \pmod{n}$, but then x would probably not be a meaningful message, so A would realize that something had been changed.

The preceding method is essentially the RSA signature scheme, which will be studied in Section 13.1.

9.7 The Public Key Concept

In 1976, Diffie and Hellman described the concept of public key cryptography, though at that time no realizations of the concept were publicly known (as mentioned in the introduction to this chapter, Clifford Cocks of the British cryptographic agency CESG had invented a secret version of RSA in 1973). In this section, we give the general theory of public key systems.

There are several implementations of public key cryptography other than RSA. In later chapters we describe three of them. One is due to ElGamal and is based on the difficulty of finding

discrete logarithms. A second is NTRU and involves lattice methods. The third is due to McEliece and uses error correcting codes. There are also public key systems based on the knapsack problem. We don't cover them in this book; some versions have been broken and they are generally suspected to be weaker than systems such as RSA and ElGamal.

A **public key cryptosystem** is built up of several components. First, there is the set M of possible messages (potential plaintexts and ciphertexts). There is also the set K of "keys." These are not exactly the encryption/decryption keys; in RSA, a key k is a triple (e, d, n) with $ed \equiv 1 \pmod{\phi(n)}$. For each key k, there

is an encryption function Ek and a decryption function Dk. Usually, Ek and Dk are assumed to map M to M, though it would be possible to have variations that allow the plaintexts and ciphertexts to come from different sets. These components must satisfy the following requirements:

1. Ek(Dk(m))=m and Dk(Ek(m))=m for every $m \in M$ and every $k \in K$.

2. For every m and every k, the values of Ek(m) and Dk(m) are easy to compute.

3. For almost every $k \in K$, if someone knows only the function Ek, it is computationally infeasible to find an algorithm to compute Dk.

4. Given $k \in K$, it is easy to find the functions Ek and Dk.

Requirement (1) says that encryption and decryption cancel each other.

Requirement (2) is needed; otherwise, efficient encryption and decryption would not be possible. Because of (4), a user can choose a secret random k from K and obtain functions Ek and Dk. Requirement (3) is what makes the system public key. Since it is difficult to determine Dk from Ek, it is possible to publish Ek without compromising the security of the system.

Let's see how RSA satisfies these requirements. The message space can be taken to be all nonnegative integers. As we mentioned previously, a key for RSA is a triple k=(e, d, n). The encryption function is

$$E_k(m) = m^e \pmod{n},$$

where we break m into blocks if $m \geq n$.

The decryption function is

$$D_k(m) = m^d \pmod{n},$$

again with m broken into blocks if needed. The functions Ek and Dk are immediately determined from knowledge of k (requirement (4)) and are easy to compute (requirement (2)). They are inverses of each other since $ed \equiv 1 \pmod{\phi(n)}$, so (1) is satisfied. If we know Ek, which means we know e and n, then we have seen that it is (probably) computationally infeasible to determine d, hence Dk. Therefore, (3) is (probably) satisfied.

Once a public key system is set up, each user generates a key k and determines Ek and Dk. The encryption function Ek is made public, while Dk is kept secret.

If there is a problem with impostors, a trusted authority can be used to distribute and verify keys.

In a symmetric system, Bob can be sure that a message that decrypts successfully must have come from Alice (who could really be a group of authorized users) or someone who has Alice's key. Only Alice has been given the key, so no one else could produce the ciphertext. However, Alice could deny sending the message since Bob could have simply encrypted the message himself. Therefore, authentication is easy (Bob knows that the message came from Alice, if he didn't forge it himself) but non-repudiation is not (see Section 1.2).

In a public key system, anyone can encrypt a message and send it to Bob, so he will have no idea where it came from. He certainly won't be able to prove it came from Alice. Therefore, more steps are needed for authentication and non-repudiation. However, these goals are easily accomplished as follows.

Alice starts with her message m and computes Ekb(Dka(m)), where ka is Alice's key and kb is Bob's key. Then Bob can decrypt using Dkb to obtain Dka(m). He uses the publicly available Eka to obtain Eka(Dka(m))=m. Bob knows that the message must have come from Alice since no one else could have computed Dka(m). For the same reason,

Alice cannot deny sending the message. Of course, all this assumes that most random “messages” are meaningless, so it is unlikely that a random string of symbols decrypts to a meaningful message unless the string was the encryption of something meaningful.

It is possible to use one-way functions with certain properties to construct a public key cryptosystem. Let f(m) be an invertible one-way function. This means f(x) is easy to compute, but, given y, it is computationally infeasible to find the unique value of x such that y=f(x). Now suppose f(x) has a **trapdoor**, which means that there is an easy way to solve y=f(x) for x, but only with some extra information known only to the designer

of the function. Moreover, it should be computationally infeasible for someone other than the designer of the function to determine this trapdoor information. If there is a very large family of one-way functions with trapdoors, they can be used to form a public key cryptosystem. Each user generates a function from the family in such a way that only that user knows the trapdoor. The user's function is then published as a public encryption algorithm. When Alice wants to send a message m to Bob, she looks up his function fb(x) and computes y=fb(m). Alice sends y to Bob. Since Bob knows the trapdoor for fb(x), he can solve y=fb(m) and thus find m.

In RSA, the functions f(x)=xe (mod n),

for appropriate n and e, form the family of one-way functions. The secret trapdoor information is the factorization of n, or, equivalently, the exponent d. In the ElGamal system (Section 10.5), the one-way function is obtained from exponentiation modulo a prime, and the trapdoor information is knowledge of a discrete log. In NTRU (Section 23.4), the trapdoor information is a pair of small polynomials. In the McEliece system (Section 24.10), the trapdoor information is an efficient way for finding the nearest codeword ("error correction") for certain linear binary codes.

9.8 Exercises

1. The ciphertext 5859 was obtained from the RSA algorithm using n=11413 and e=7467. Using the factorization 11413=101·113, find the plaintext.

2. Bob sets up a budget RSA cryptosystem. He chooses p=23 and q=19 and computes n=437=pq. He chooses the encryption exponent to be e=397. Alice sends Bob the ciphertext c=123. What is the plaintext? (You know p, q, and e).

3. Suppose your RSA modulus is n=55=5×11 and your encryption exponent is e=3.

    1. Find the decryption exponent d.

    2. Assume that gcd(m, 55)=1. Show that if $c \equiv m^3 \pmod{55}$ is the ciphertext, then the plaintext is $m \equiv c^d \pmod{55}$. Do not quote the fact that RSA decryption works. That is what you are showing in this specific case.

4. Bob's RSA modulus is 979=11×89 and his encryption exponent is e=587. Alice sends him the

ciphertext c=10. What is the plaintext?

5. The ciphertext 62 was obtained using RSA with n=667 and e=3. You know that the plaintext is either 9 or 10. Determine which it is without factoring n.

6. Alice and Bob are trying to use RSA, but Bob knows only one large prime, namely p=1093. He sends n=p=1093 and e=361 to Alice. She encrypts her message m as $c \equiv m^e \pmod{n}$. Eve intercepts c and decrypts using the congruence $m \equiv c^d \pmod{n}$. What value of d should Eve use? Your answer should be an actual number. You may assume that Eve knows n and e, and she knows that n is prime.

7. Suppose you encrypt messages m by computing $c \equiv m^3 \pmod{101}$. How do you decrypt? (That is, you want a decryption exponent d such that $c^d \equiv m \pmod{101}$; note that 101 is prime.)

8. Bob knows that if an RSA modulus can be factored, then the system has bad security. Therefore, he chooses a modulus that cannot be factored, namely the prime n=131303. He chooses his encryption exponent to be e=13, and he encrypts a message m as $c \equiv m^{13} \pmod{n}$. The decryption method is to compute $m \equiv c^d \pmod{n}$ for some d. Find d. (Hint: Fermat's theorem)

9. Let p be a large prime. Suppose you encrypt a message x by computing $y \equiv x^e \pmod{p}$ for some (suitably chosen) encryption exponent e. How do you find a decryption exponent d such that $y^d \equiv x \pmod{p}$?

10. Bob decides to test his new RSA cryptosystem. He has RSA modulus n=pq and encryption exponent e. His message is m. He sends $c \equiv m^e \pmod{n}$ to himself. Then, just for fun, he also sends $c' \equiv (m+q)^e \pmod{n}$ to himself. Eve knows n and e, and intercepts both c and c′. She guesses what Bob has done. How can she find the factorization of n? (Hint: Show that $c \equiv c' \pmod{q}$ but not mod p. What is gcd(c−c′, n)?)

11. Let n be the product of two large primes. Alice wants to send a message m to Bob, where gcd(m, n)=1. Alice and Bob choose integers a and b relatively prime to $\phi(n)$. Alice computes $c \equiv m^a \pmod{n}$ and sends c to Bob. Bob computes $d \equiv c^b \pmod{n}$ and sends d back to Alice. Since Alice knows a, she finds a_1 such that $aa_1 \equiv 1 \pmod{\phi(n)}$. Then she computes $e \equiv d^{a_1} \pmod{n}$ and sends e to Bob. Explain what Bob must now do to obtain m, and show that this works. (Remark: In this protocol, the prime factors of n do not need to be kept secret. Instead, the security depends on keeping a, b secret. The present protocol is a less efficient version of the three-pass protocol from

Section 3.5.)

12. A bank in Alice Springs (Australia), also known as Alice, wants to send a lot of financial data to the Bank of Baltimore, also known as Bob. They want to use AES, but they do not share a common key. All of their communications will be on public airwaves. Describe how Alice and Bob can accomplish this using RSA.

13. Naive Nelson uses RSA to receive a single ciphertext c, corresponding to the message m. His public modulus is n and his public encryption exponent is e. Since he feels guilty that his system was used only once, he agrees to decrypt any ciphertext that someone sends him, as long as it is not c, and return the answer to that person. Evil Eve sends him the ciphertext $2^e c \pmod{n}$. Show how this allows Eve to find m.

14. Eve loves to do double encryption. She starts with a message m. First, she encrypts it twice with a one-time pad (the same one each time). Then she encrypts the result twice using a Vigenère cipher with key NANANA. Finally, she encrypts twice with RSA using modulus $n=pq=7919\times17389$ and exponent $e=66909025$. It happens that $e^2 \equiv 1 \pmod{(p-1)(q-1)}$. Show that the final result of all this encryption is the original plaintext. Explain your answer fully. Simply saying something like

"decryption is the same as encryption" is not enough. You must explain why.

15. In order to increase security, Bob chooses n and two encryption exponents e1, e2. He asks Alice to encrypt her message m to him by first computing $c1 \equiv me1 \pmod{n}$, then encrypting c1 to get $c2 \equiv c1e2 \pmod{n}$. Alice then sends c2 to Bob. Does this double encryption increase security over single encryption? Why or why not?

16.

    1. Eve thinks that she has a great strategy for breaking RSA that uses a modulus n that is the product of two 300-digit primes. She decides to make a list of all 300-digit primes and then divide each of them into n until she factors n. Why won't this strategy work?

    2. Eve has another strategy. She will make a list of all m with $0 \leq m < 10600$, encrypt each m, and store them in a database. Suppose Eve has a superfast computer that can encrypt 1050 plaintexts per second (this is, of course, well beyond the speed of any existing technology). How many years will it take Eve to compute all 10600 encryptions? (There are approximately 3×107 seconds in a

year.)

17. The exponents e=1 and e=2 should not be used in RSA. Why?

18. Alice is trying to factor n=57677. She notices that $12344 \equiv 1 \pmod{n}$ and that $12342 \equiv 23154 \pmod{n}$. How does she use this information to factor n? Describe the steps but do not actually factor n.

19. Let p and q be distinct odd primes, and let n=pq. Suppose that the integer x satisfies gcd(x, pq)=1.

    1. Show that $x^{12\phi(n)} \equiv 1 \pmod{p}$ and $x^{12\phi(n)} \equiv 1 \pmod{q}$.

    2. Use (a) to show that $x^{12\phi(n)} \equiv 1 \pmod{n}$.

    3. Use (b) to show that if $ed \equiv 1 \pmod{12\phi(n)}$ then $x^{ed} \equiv x \pmod{n}$. (This shows that we could work with $12\phi(n)$ instead of $\phi(n)$ in RSA. In fact, we could also use the least common multiple of $p-1$ and $q-1$ in place of $\phi(n)$, by similar reasoning.)

20. Alice uses RSA with n=27046456501 and e=3.

Her ciphertext is c=1860867. Eve notices that $c^{14} \equiv 1 \pmod{n}$.

1. Show that $m^{14} \equiv 1 \pmod{n}$, where m is the plaintext.

2. Explicitly find an exponent f such that $c^f \equiv m \pmod{n}$. (Hint: You do not need to factor n to find f. Look at the proof that RSA decryption works. The only property of $\phi(n)$ that is used is that $m^{\phi(n)} \equiv 1$.)

21. Suppose that there are two users on a network. Let their RSA moduli be n1 and n2, with n1 not equal to n2. If you are told that n1 and n2 are not relatively prime, how would you break their systems?

22. Huey, Dewey, and Louie ask their uncle Donald, "Is n=19887974881 prime or composite?" Donald replies, "Yes." Therefore, they decide to obtain more information on their own.

    1. Huey computes $13^{(n-1)} \equiv 16739180549 \pmod{n}$. What does he conclude?

    2. Dewey computes $7^{n-1} \equiv 1 \pmod{n}$, and he does this by computing

$7^{(n-1)/32} \equiv 1992941816 \pmod{n}$ $7^{(n-1)/16} \equiv 198877306197^{(n-1)/8} \equiv 17^{(n-1)/4} \equiv 7^{(n-1)/2} \equiv 7^{(n-1)} \equiv 1$.

What information can Dewey obtain from his calculation that Huey does not obtain?

3. Louie notices that $198579306552 \equiv 1232 \pmod{n}$. What information can Louie compute? (In parts (b) and (c), you do not need to do the calculations, but you should indicate what calculations need to be done.)

23. You are trying to factor n=642401. Suppose you discover that

$$5161072 \equiv 7 \pmod{n}$$

and that

$$1877222 \equiv 22 \cdot 7 \pmod{n}.$$

Use this information to factor n.

24. Suppose you know that $79612 \equiv 72 \pmod{8051}$. Use this information to factor 8051.

25. Suppose you discover that

$$8805252 \equiv 2, \quad 20572022 \equiv 3,$$

$6485812 \equiv 6, 6686762 \equiv 77 \pmod{2288233}$.

How would you use this information to factor 2288233? Explain what steps you would do, but do not perform the numerical calculations.

26. Suppose you want to factor an integer n. You have found some integers x1, x2, x3, x4 such that

$x12 \equiv 2 \cdot 3 \cdot 7 \pmod{n}$, $x22 \equiv 3 \cdot 5 \cdot 7 \pmod{n}$ $x32 \equiv 39 \pmod{n}$, $x42 \equiv 2 \cdot 7 \pmod{n}$.

Describe how you might be able to use this information to factor n? Why might the method fail?

27. Suppose you have two distinct large primes p and q. Explain how you can find an integer x such that

$x2 \equiv 49 \pmod{pq}$, $x \equiv \pm 7 \pmod{pq}$.

(Hint: Use the Chinese Remainder Theorem to find four solutions to $x2 \equiv 49 \pmod{pq}$.)

28. You are told that

$5945 \equiv 1768 \pmod{1891}$, $51890 \equiv 1 \pmod{1891}$.

Use this information to factor 1891.

29. Suppose n is a large odd number. You calculate $2^{(n-1)/2} \equiv k \pmod{n}$, where k is some integer with $k \equiv \pm 1 \pmod{n}$.

1. Suppose $k^2 \equiv 1 \pmod{n}$. Explain why this implies that n is not prime.

2. Suppose $k^2 \equiv 1 \pmod{n}$. Explain how you can use this information to factor n.

30.

1. Bob is trying to set up an RSA cryptosystem. He chooses $n=pq$ and e, as usual. By encrypting messages six times, Eve guesses that $e^6 \equiv 1 \pmod{(p-1)(q-1)}$. If this is the case, what is the decryption exponent d? (That is, give a formula for d in terms of the parameters n and e that allows Eve to compute d.)

2. Bob tries again, with a new n and e. Alice computes $c \equiv m^e \pmod{n}$. Eve sees no way to guess the decryption exponent d this time. Knowing that if she finds d she will have to do modular exponentiation, Eve starts computing successive squares: $c, c^2, c^4, c^8, \cdots \pmod{n}$. She notices that $c^{32} \equiv 1 \pmod{n}$ and realizes that this means that $m^{32} \equiv 1 \pmod{n}$. If $e=53$, what is a value for d that will decrypt the ciphertext? Prove that this value works.

31. Suppose two users Alice and Bob have the same RSA modulus n and suppose that their encryption exponents eA and eB are relatively prime. Charles wants to send the message m to Alice and Bob, so he encrypts to get $cA \equiv meA$ and $cB \equiv meB \pmod{n}$. Show how Eve can find m if she intercepts cA and cB.

32. Bob finally sets up a very secure RSA system. In fact, it is so secure that he decides to tell Alice one of the prime factors of n; call it p. Being no dummy, he does not take the message as p, but instead uses a shift cipher to hide the prime in the plaintext m=p+1, and then does an RSA encryption to obtain $c \equiv me \pmod{n}$. He then sends c to Alice. Eve intercepts c, n and e, and she knows that Bob has encrypted p this way. Explain how she obtains p and q quickly. (Hint: How does c differ from the result of encrypting the simple plaintext "1" ?)

33. Suppose Alice uses the RSA method as follows. She starts with a message consisting of several letters, and assigns a=1, b=2, …, z=26. She then encrypts each letter separately. For example, if her message is cat, she calculates $3e \pmod{n}$, $1e \pmod{n}$, and $20e \pmod{n}$. Then she sends the encrypted message to Bob. Explain how Eve can find the message without factoring n. In particular, suppose n=8881 and e=13. Eve intercepts the message

4461794201520153603.

Find the message without factoring 8881.

34. Let n=3837523. Bob Square Messages sends and receives only messages m such that m is a square mod n and gcd(m, n)=1. It can be shown that $m^{958230} \equiv 1 \pmod{n}$ for such messages (even though $\phi(n) \neq 958230$). Bob chooses d and e satisfying $de \equiv 1 \pmod{958230}$. Show that if Alice sends Bob a ciphertext $c \equiv m^e \pmod{n}$ (where m is a square mod n, and gcd(m, n)=1), then Bob can decrypt by computing $c^d \pmod{n}$. Explain your reasoning.

35. Show that if $x^2 \equiv y^2 \pmod{n}$ and $x \not\equiv \pm y \pmod{n}$, then gcd(x+y, n) is a nontrivial factor of n.

36. Bob's RSA system uses n=2776099 and e=421. Alice encrypts the message 2718 and sends the ciphertext to Bob. Unfortunately (for Alice), $271^{810} \equiv 1 \pmod{n}$. Show that Alice's ciphertext is the same as the plaintext. (Do not factor n. Do not compute $m^{421} \pmod{n}$ without using the extra information that $271^{810} \equiv 1$. Do not claim that $\phi(2776099=10$; it doesn't.)

37. Let n=pq be the product of two distinct primes.

    1. Let m be a multiple of $\phi(n)$. Show that if

gcd(a, n)=1, then $am \equiv 1 \pmod{p}$ and $\pmod{q}$.

2. Suppose m is as in part (a), and let a be arbitrary (possibly $\gcd(a, n) \neq 1$). Show that $am+1 \equiv a \pmod{p}$ and $\pmod{q}$.

3. Let e and d be encryption and decryption exponents for RSA with modulus n. Show that $aed \equiv a \pmod{n}$ for all a. This shows that we do not need to assume gcd(a, n)=1 in order to use RSA.

4. If p and q are large, why is it likely that gcd(a, n)=1 for a randomly chosen a?

38. Alice and Bob are celebrating Pi Day. Alice calculates $230392 \equiv 314 \pmod{79927}$ and Bob calculates $351182 \equiv 314 \pmod{79927}$.

    1. Use this information to factor 79927. (You should show how to use the information. Do not do the calculation. The answer is not "Put the number in the computer and then relax for 10 minutes.")

    2. Given that $79927=257\times311$ and that $1662 \equiv 314 \pmod{257}$ and $252 \equiv 314 \pmod{311}$, how would you produce the

numbers 23039 and 35118 in part (a)? You do not need to do the calculations, but you should state which congruences are being solved and what theorems are being used.

3. Now that Eve knows that $79927=257\times311$, she wants to find an $x \equiv \pm17 \pmod{79927}$ such that $x2 \equiv 172 \pmod{79927}$. Explain how to accomplish this. Say what the method is. Do not do the calculation.

39. Suppose $n=pqr$ is the product of three distinct primes. How would an RSA-type scheme work in this case? In particular, what relation would e and d satisfy?

    Note: There does not seem to be any advantage in using three primes instead of two. The running times of some factorization methods depend on the size of the smallest prime factor. Therefore, if three primes are used, the size of n must be increased in order to achieve the same level of security as obtained with two primes.

40. Suppose Bob's public key is $n=21816061489508751380 77$ and he has $e=7$ as his encryption exponent. Alice encrypts the

message *hi eve* = 080900052205=m. By chance, the message m satisfies $m3 \equiv 1 \pmod{n}$. If Eve intercepts the ciphertext, how can Eve read the message without factoring n?

41. Let p=7919 and q=17389. Let e=66909025. A calculation shows that $e2 \equiv 1 \pmod{(p-1)(q-1)}$. Alice decides to encrypt the message m=12345 using RSA with modulus n=pq and exponent e. Since she wants the encryption to be very secure, she encrypts the ciphertext, again using n and e (so she has double encrypted the original plaintext). What is the final ciphertext that she sends? Justify your answer without using a calculator.

42. You are told that $71722 \equiv 602 \pmod{14351}$. Use this information to factor 14351. You must use this information and you must give all steps of the computation (that is, give the steps you use if you are doing it completely without a calculator).

43.

    1. Show that if gcd(e, 24)=1, then $e2 \equiv 1 \pmod{24}$.

    2. Show that if n=35 is used as an RSA modulus, then the encryption exponent e always equals the decryption exponent d.

44.

1. The exponent e=3 has sometimes been used for RSA because it makes encryption fast. Suppose that Alice is encrypting two messages, m0=m and m1=m+1 (for example, these could be two messages that contain a counter variable that increments). Eve does not know m but knows that the two plaintexts differ by 1. Let $c0 \equiv m0^3$ and $c1 \equiv m1^3 \bmod n$. Show that if Eve knows c0 and c1, she can recover m. (Hint: Compute $(c1+2c0-1)/(c1-c0+2)$.)

2. Suppose that m0=m and m1=m+b for some publicly known b. Modify the technique of part (a) so that Eve can recover m from the ciphertexts c0 and c1.

45. Your opponent uses RSA with n=pq and encryption exponent e and encrypts a message m. This yields the ciphertext $c \equiv m^e \pmod n$. A spy tells you that, for this message, $m^{12345} \equiv 1 \pmod n$. Describe how to determine m. Note that you do not know p, q, $\phi(n)$, or the secret decryption exponent d. However, you should find a decryption exponent that works for this particular ciphertext. Moreover, explain carefully why your decryption works (your explanation must include how the spy's information is used). For simplicity,

assume that gcd(12345, e)=1.

46.

1. Show that if gcd(b, 1729)=1 then $b^{1728} \equiv 1 \pmod{1729}$. You may use the fact that $1729=7\cdot13\cdot19$.

2. By part (a), you know that $2^{1728} \equiv 1 \pmod{1729}$. When checking this result, you compute $2^{54} \equiv 1065 \pmod{1729}$ and $2^{108} \equiv 1 \pmod{1729}$. Use this information to find a nontrivial factor of 1729.

47. Suppose you are using RSA (with modulus n=pq and encrypting exponent e), but you decide to restrict your messages to numbers m satisfying $m^{1000} \equiv 1 \pmod{n}$.

1. Show that if d satisfies $de \equiv 1 \pmod{1000}$, then d works as a decryption exponent for these messages.

2. Assume that both p and q are congruent to 1 mod 1000. Determine how many messages satisfy $m^{1000} \equiv 1 \pmod{n}$. You may assume and use the fact that $m^{1000} \equiv 1 \pmod{r}$ has 1000 solutions when r is a prime congruent to 1 mod 1000.

48. You may assume the fact that $m^{270300} \equiv 1 \pmod{1113121}$ for all m with gcd(m, 1113121)=1. Let e and d satisfy $ed \equiv 1 \pmod{270300}$, and suppose that m is a message such that $0<m<1113121$ and gcd(m, 1113121)=1. Encrypt m as $c \equiv m^e \pmod{1113121}$. Show that $m \equiv c^d \pmod{1113121}$. Show explicitly how you use the fact that $ed \equiv 1 \pmod{270300}$ and the fact that $m^{270300} \equiv 1 \pmod{1113121}$. (Note: $\phi(1113121) \neq 270300$, so Euler's theorem does not apply.)

49. Suppose Bob's encryption company produces two machines, A and B, both of which are supposed to be implementations of RSA using the same modulus n=pq for some unknown primes p and q. Both machines also use the same encryption exponent e. Each machine receives a message m and outputs a ciphertext that is supposed to be $m^e \pmod{n}$. Machine A always produces the correct output c. However, Machine B, because of implementation and hardware errors, always outputs a ciphertext $c' \pmod{n}$ such that $c' \equiv m^e \pmod{p}$ and $c' \equiv m^{e+1} \pmod{q}$. How could you use machines A and B to find p and q? (See Computer Problem 11 for a discussion of how such a situation could arise.) (Hint: $c \equiv c' \pmod{p}$ but not mod q. What is $\gcd(c-c', n)$?)

50. Alice and Bob play the following game. They

choose a large odd integer n and write $n-1=2^k m$ with m odd. Alice then chooses a random integer $r \not\equiv \pm 1 \pmod{n}$ with $\gcd(r, n)=1$. Bob computes $x1 \equiv r^m \pmod{n}$. Then Alice computes $x2 \equiv x1^2 \pmod{n}$. Then Bob computes $x3 \equiv x2^2 \pmod{n}$, Alice computes $x4 \equiv x3^2 \pmod{n}$, etc. They stop if someone gets $\pm 1 \pmod{n}$, and the person who gets ± 1 wins.

1. Show that if n is prime, the game eventually stops.

2. Suppose n is the product of two distinct primes and Alice knows this factorization. Show how Alice can choose r so that she wins on her first play. That is, $x1 \not\equiv \pm 1 \pmod{n}$ but $x2 \equiv \pm 1 \pmod{n}$.

51.

1. Suppose Alice wants to send a short message m but wants to prevent the short message attack of Section 9.2. She tells Bob that she is adjoining 100 zeros at the end of her plaintext, so she is using $m1=10^{100}m$ as the plaintext and sending $c1 \equiv m1^e$. If Eve knows that Alice is doing this, how can Eve modify the short plaintext attack and possibly find the

plaintext?

2. Suppose Alice realizes that the method of part (a) does not provide security, so instead she makes the plaintext longer by repeating it two times: m||m (where x||y means we write the digits of x followed by the digits of y to obtain a longer number). If Eve knows that Alice is doing this, how can Eve modify the short plaintext attack and possibly find the plaintext? Assume that Eve knows the length of m. (Hint: Express m||m as a multiple of m.)

52. This exercise provides some of the details of how the quadratic sieve obtains the relations that are used to factor a large odd integer n. Let s be the smallest integer greater than the square root of n and let $f(x)=(x+s)2-n$. Let the factor base B consist of the primes up to some bound B. We want to find squares that are congruent mod n to a product of primes in B. One way to do this is to find values of f(x) that are products of primes in B. We'll search over a range $0 \le x \le A$, for some A.

    1. Suppose $0 \le x<(2-1)n-1$. Show that $0 \le f(x)<n$, so f(x) (mod n) is simply f(x). (Hint: Show that $x+s<x+n+1<2n$.)

Henceforth, we'll assume that $A<(2-1)n-1$, so the values of x that we consider have $f(x)<n$.

2. Let p be a prime in B. Show that if there exists an integer x with f(x) divisible by p, then n is a square mod p. This shows that we may discard those primes in B for which n is not a square mod p. Henceforth, we will assume that such primes have been discarded.

3. Let $p \in B$ be such that n is a square mod p. Show that if p is odd, and $p|n$, then there are exactly two values of x mod p such that $f(x) \equiv 0 \pmod{p}$. Call these values $x_{p,1}$ and $x_{p,2}$. (Note: In the unlikely case that $p|n$, we have found a factor, which was the goal.)

4. For each x with $0 \le x \le A$, initialize a register with value $\log f(x)$. For each prime $p \in B$, subtract $\log p$ from the registers of those x with $x \equiv x_{p,1}$ or $x_{p,2} \pmod{p}$. (Remark: This is the "sieving" part of the quadratic sieve.) Show that if f(x) (with $0 \le x \le A$) is a product of distinct primes in B, then the register for x becomes 0 at the end of this process.

5. Explain why it is likely that if f(x) (with $0 \le x \le A$) is a product of (possibly nondistinct) primes in B, then the final result for the register for x is small (compared to the register for an x such that f(x) has a prime factor not in B).

6. Why is the procedure of part (d) faster than trial division of each f(x) by each element of B, and why does the algorithm subtract logp rather than dividing f(x) by p?

In practice, the sieve also takes into account solutions to $f(x) \equiv 0$ mod some powers of small primes in B. After the sieving process is complete, the registers with small entries are checked to see which correspond to f(x) being a product of primes from B. These give the relations "square $\equiv$ product of primes in B mod n" that are used to factor n.

53. Bob chooses n=pq to be the product of two large primes p, q such that gcd(pq, (p−1)(q−1))=1.

1. Show that $\phi(n2)=n\phi(n)$ (this is true for any positive n).

2. Let $k \ge 0$. Show that $(1+n)k \equiv 1+kn$ (mod n2) (this is true for any positive n).

3. Alice has message represented as m (mod n). She encrypts m by first choosing a random integer r with gcd(r, n)=1. She then computes

   $$c \equiv (1+n)^m r^n \pmod{n^2}.$$

4. Bob decrypts by computing $m' \equiv c^{\phi(n)} \pmod{n^2}$. Show that $m' \equiv 1+\phi(n)mn \pmod{n^2}$. Therefore, Bob can recover the message by computing $(m'-1)/n$ and then dividing by $\phi(n) \pmod{n}$.

5. Let E(m) and D(c) denote encryption and decryption via the method in parts (c) and (d). Show that

   $$D(E(m_1)E(m_2)) \equiv D(E(m_1+m_2)) \pmod{n}. \quad (9.1)$$

   Note: The encryptions probably use different values of the random number r, so there is more than one possible encryption of a message m. Part (e) says that, no matter what choices are made for r, the decryption of E(m1)E(m2) is the same as the decryption of E(m1+m2).

The preceding is called the **Paillier cryptosystem**. (Equation 9.1) says that is is possible to do addition on the encrypted messages without

knowing the messages. For many years, a goal was to design a cryptosystem where both addition and multiplication could be done on the encrypted messages. This property is called **homomorphic encryption**, and the first such system was designed by Gentry in 2009. Current research aims at designing systems that can be used in practice.

54. One possible application of the Paillier cryptosystem from the previous exercise is to electronic voting (but, as we'll see, modifications are needed in order to make it secure). Bob, who is the trusted authority, sets up the system. Each voter uses m=0 for NO and m=1 for YES. The voters encrypt their ms and send the ciphertexts to Bob.

    1. How does Bob determine how many YES and how many NO votes without decrypting the individual votes?

    2. Suppose an overzealous and not very honest voter wants to increase the number of YES votes. How is this accomplished?

    3. Suppose someone else wants to increase the number of NO votes. How can this be done?

55. Here is a 3-person encryption scheme based on the same principles as RSA. A trusted entity chooses two large distinct primes p and q and computes $n=pq$, then chooses three integers k1, k2, k3 with $k1k2k3 \equiv 1 \pmod{(p-1)(q-1)}$. Alice, Bob, and Carla are given the following keys:

    Alice: (n, k1, k2), Bob: (n, k2, k3), Carla: (n, k1, k3).

    1. Alice has a message that she wants to send to both Bob and Carla. How can Alice encrypt the message so that both of them can read decrypt it?

    2. Alice has a message that she wants to send only to Carla. How can Alice encrypt the message so that Carla can decrypt it but Bob cannot decrypt it?

9.9 Computer Problems

1.

    1. Paul Revere's friend in a tower at MIT says he'll send the message *one* if (the British are coming) by land and *two* if by sea. Since they know that RSA will be invented in the Boston area, they decide that the message should be encrypted using RSA with n=712446816787 and e=6551. Paul Revere receives the ciphertext 273095689186. What was the plaintext? Answer this without factoring n.

    2. What could Paul Revere's friend have done so that we couldn't guess which message was encrypted? (See the end of Subsection 9.2.2.)

2. In an RSA cryptosystem, suppose you know n=718548065973745507, e=3449, and d=543546506135745129. Factor n using the ar ≡ 1 method of Subsection 9.4.2.

3. Choose two 30-digit primes p and q and an

encryption exponent e. Encrypt each of the plaintexts *cat, bat, hat, encyclopedia, antidisestablishmentarianism*. Can you tell from looking at the ciphertexts that the first three plaintexts differ in only one letter or that the last two plaintexts are much longer than the first three?

4. Factor 6182400071090270 21 by the p−1 method.

5. Factor 8834884587090814646372459890377418962766907 by the p−1 method. (The number is stored in the downloadable computer files (`bit.ly/2JbcS6p`) as *n1*.)

6. Let n=537069139875071. Suppose you know that

   $$859753244431662 \equiv 4624361062612 \pmod{n}.$$

   Factor n.

7. Let n=985739879·1388749507. Find x and y with $x^2 \equiv y^2 \pmod{n}$ but $x \not\equiv \pm y \pmod{n}$.

8. 1. Suppose you know that

      $$333352 \equiv 6707050932 \pmod{670726081}.$$

      Use this information to factor 670726081.

2. Suppose you know that
   $32 \equiv 6707260782 \pmod{670726081}$.
   Why won't this information help you to factor 670726081?

9. Suppose you know that

   $2958230 \equiv 1488665 \pmod{3837523}$ $21916460 \equiv 1 \pmod{3837523}$.

   How would you use this information to factor 3837523? Note that the exponent 1916460 is twice the exponent 958230.

   Alice and Bob have the same RSA modulus n, given to them by some central authority (who does not tell them the factorization of n). Alice has encryption and decryption exponents eA and dA, and Bob has eB and dB. As usual, eA and eB are public and dA and dB are private.

10.
    1. Suppose the primes p and q used in the RSA algorithm are consecutive primes. How would you factor n=pq?

    2. The ciphertext 10787770728 was encrypted using e=113 and n=10993522499. The factors p and q of n were chosen so that $q-p=2$. Decrypt the

message.

3. The following ciphertext c was encrypted mod n using the exponent e:

   n=152415787501905985701881832150835089037858867

   The prime factors p and q of n are consecutive primes. Decrypt the message. (The number n is stored in the downloadable computer files (`bit.ly/2JbcS6p`) as *naive*, and c is stored as *cnaive*.) Note: In Mathematica®, the command **Round[N[Sqrt[n],50]]** evaluates the square root of n to 50 decimal places and then rounds to the nearest integer. In Maple, first use the command **Digits:=50** to obtain 50-digit accuracy, then use the command **round(sqrt(n*1.))** to change n to a decimal number, take its square root, and round to the nearest integer. In MATLAB, use the command `digits(50);round(vpa(sqrt(n')))`.

11. Let p=123456791, q=987654323, and e=127. Let the message be m=14152019010605.

    1. Compute $m^e \pmod{p}$ and $m^e \pmod{q}$; then use the Chinese remainder theorem

to combine these to get $c \equiv m^e \pmod{pq}$.

2. Change one digit of $m^e \pmod{p}$ (for example, this could be caused by some radiation). Now combine this with $m^e \pmod{q}$ to get an incorrect value f for $m^e \pmod{pq}$. Compute $\gcd(c-f, pq)$. Why does this factor pq?

The method of (a) for computing $m^e \pmod{pq}$ is attractive since it does not require as large multiprecision arithmetic as working directly mod pq. However, as part (b) shows, if an attacker can cause an occasional bit to fail, then pq can be factored.

12. Suppose that $p=76543692179$, $q=343434343453$, and $e=457$. The ciphertext $c \equiv m^e \pmod{pq}$ is transmitted, but an error occurs during transmission. The received ciphertext is $230432932801693694719 5$. The receiver is able to determine that the digits received are correct but that last digit is missing. Determine the missing digit and decrypt the message.

13. Test 38200901201 for primality using the Miller-Rabin test with $a=2$. Then test using $a=3$. Note that the first test says that 38200901201 is probably prime, while the second test says that it is

composite. A composite number such as 38200901201 that passes the Miller-Rabin test for a number a is called a **strong a-pseudoprime**.

14. There are three users with pairwise relatively prime moduli n1, n2, n3. Suppose that their encryption exponents are all e=3. The same message m is sent to each of them and you intercept the ciphertexts $c_i \equiv m^3 \pmod{n_i}$ for i=1, 2, 3.

    1. Show that $0 \leq m^3 < n_1 n_2 n_3$.

    2. Show how to use the Chinese remainder theorem to find m^3 (as an exact integer, *not* only as $m^3 \pmod{n_1 n_2 n_3}$) and therefore m. Do this without factoring.

    3. Suppose that

       n1=2469247531693,
       n2=11111502225583, n3=44444222221411

       and the corresponding ciphertexts are

       359335245251,
       10436363975495, 5135984059593.

       These were all encrypted using e=3. Find the message.

15.

1. Choose a 10-digit prime p and and 11-digit prime q. Form n=pq.

2. Let the encryption exponent be e=65537. Write a program that computes the RSA encryptions of all plaintexts m with $1 \le m < 104$. (Do not store or display the results.) The computer probably did this almost instantaneously.

3. Modify your program in (b) so that it computes the encryptions of all m with $1 \le m < 108$ and time how long this takes (if this takes too long, use 107; if it's too fast to time, use 109).

4. Using your timing from (c), estimate how long it will take to encrypt all m with $1 \le m < n$ (a year is approximately 3×107 seconds).

Even this small example shows that it is impractical to make a database of all encryptions in order to attack RSA.

Chapter 10 Discrete Logarithms

10.1 Discrete Logarithms

In the RSA algorithm, we saw how the difficulty of factoring yields useful cryptosystems. There is another number theory problem, namely discrete logarithms, that has similar applications.

Fix a prime p. Let α and β be nonzero integers mod p and suppose

$$\beta \equiv \alpha x (\text{mod } p)$$

The problem of finding x is called the **discrete logarithm problem**. If n is the smallest positive integer such that

$\alpha n \equiv 1 \pmod{p}$, we may assume $0 \leq x < n$, and then we denote

$$x = L_\alpha(\beta)$$

and call it the discrete log of β with respect to α (the prime p is omitted from the notation).

For example, let $p=11$ and let $\alpha=2$. Since $26 \equiv 9 \pmod{11}$, we have $L_2(9)=6$. Of course, $26 \equiv 216 \equiv 226 \equiv 9 \pmod{11}$, so we could consider taking any one of 6, 16, 26 as the discrete logarithm. But we fix the value by taking the smallest nonnegative value, namely 6. Note that we could have defined the discrete logarithm in this case to be the congruence class 6 mod 10. In some ways, this would be more natural, but

there are applications where it is convenient to have a number, not just a congruence class.

Often, α is taken to be a primitive root mod p, which means that every β is a power of α (mod p). If α is not a primitive root, then the discrete logarithm will not be defined for certain values of β.

Given a prime p, it is fairly easy to find a primitive root in many cases. See Exercise 54 in Chapter 3.

The discrete log behaves in many ways like the usual logarithm. In particular, if α is a primitive root mod p, then

$$L_\alpha(\beta_1\beta_2) \equiv L_\alpha(\beta_1)+L_\alpha(\beta_2) \pmod{p-1}$$

(see Exercise 6).

When p is small, it is easy to compute discrete logs by exhaustive search through all possible exponents. However, when p is large this is not feasible. We give some ways of attacking discrete log problems later. However, it is believed that discrete logs are hard to compute in general. This assumption is the basis of several cryptosystems.

The size of the largest primes for which discrete logs can be computed has usually been approximately the same size as the size of largest integers that could be factored (both of these refer to computations that would work for arbitrary numbers of these sizes; special

choices of integers will succumb to special techniques, and thus discrete log computations and factorizations work for much larger specially chosen numbers). Compare Table 10.1 with Table 9.2 in Chapter 9.

Table 10.1 Discrete Log Records

Year	Number of Digits of p
2001	120
2005	130
2007	160
2014	180
2016	232

A function f(x) is called a **one-way function** if f(x) is easy to compute, but, given y, it is computationally infeasible to find x with f(x)=y. Modular exponentiation is probably an example

of such a function. It is easy to compute $\alpha x \pmod{p}$, but solving $\alpha x \equiv \beta$ for x is probably hard. Multiplication of large primes can also be regarded as a (probable) one-way function: It is easy to multiply primes but difficult to factor the result to recover the primes. One-way functions have many cryptographic uses.

10.2 Computing Discrete Logs

In this section, we present some methods for computing discrete logarithms. A method based on the birthday attack is discussed in Subsection 12.1.1.

For simplicity, take α to be a primitive root mod p, so $p-1$ is the smallest positive exponent n such that $\alpha^n \equiv 1 \pmod{p}$. This implies that

$$\alpha^{m1} \equiv \alpha^{m2} \pmod{p} \Leftrightarrow m1 \equiv m2 \pmod{p-1}.$$

Assume that

$$\beta \equiv \alpha^x,\ 0 \le x < p-1.$$

We want to find x.

First, it's easy to determine x (mod 2). Note that

$$(\alpha^{(p-1)/2})^2 \equiv \alpha^{p-1} \equiv 1 \pmod{p},$$

so $\alpha^{(p-1)/2} \equiv \pm 1 \pmod{p}$ (see Exercise 15 in Chapter 3). However, $p-1$ is assumed to be the smallest exponent to yield +1, so we must have

$$\alpha^{(p-1)/2} \equiv -1 \pmod{p}.$$

Starting with $\beta \equiv \alpha^x \pmod{p}$, raise both sides to the $(p-1)/2$ power to obtain

$$\beta^{(p-1)/2} \equiv \alpha^{x(p-1)/2} \equiv (-1)^x \pmod{p}.$$

Therefore, if $\beta^{(p-1)/2} \equiv +1$, then x is even; otherwise, x is odd.

Example

Suppose we want to solve
$2x \equiv 9 \pmod{11}$. Since

$$\beta^{(p-1)/2} \equiv 9^5 \equiv 1 \pmod{11},$$

we must have x even. In fact, x=6, as we saw previously.

10.2.1 The Pohlig-Hellman Algorithm

The preceding idea was extended by Pohlig and Hellman to give an algorithm to compute discrete logs when p−1 has only small prime factors. Suppose

$$p-1 = \prod_i q_i^{r_i}$$

is the factorization of p−1 into primes. Let q^r be one of the factors. We'll compute $L_\alpha(\beta) \pmod{q^r}$. If this can be done for each $q_i^{r_i}$, the answers can be

recombined using the Chinese remainder theorem to find the discrete logarithm.

Write

$$x=x0+x1q+x2q2+\cdots \text{ with } 0 \le xi \le q-1.$$

We'll determine the coefficients x0, x1, …xr−1 successively, and thus obtain x mod qr. Note that

$$xp-1q=x0p-1q+(p-1)(x1+x2q+x3q2+\cdots)=x0p-1q+(p-1)n,$$

where n is an integer. Starting with $\beta \equiv \alpha x$, raise both sides to the (p−1)/q power to obtain

$$\beta(p-1)/q \equiv \alpha x(p-1)/q \equiv \alpha x0(p-1)/q(\alpha p-1)n \equiv \alpha x0(p-1)/q(\text{mod } p).$$

The last congruence is a consequence of Fermat's theorem: $\alpha p-1 \equiv 1 \pmod p$. To find x0, simply look at the powers

$\alpha^{k(p-1)/q} \pmod p$, $k=0, 1, 2, \ldots, q-1$,

until one of them yields $\beta^{(p-1)/q}$. Then $x_0=k$. Note that since $\alpha^{m_1} \equiv \alpha^{m_2} \Leftrightarrow m_1 \equiv m_2 \pmod{p-1}$, and since the exponents $k(p-1)/q$ are distinct mod $p-1$, there is a unique k that yields the answer.

An extension of this idea yields the remaining coefficients. Assume that $q^2 | p-1$. Let

$$\beta_1 \equiv \beta\alpha^{-x_0} \equiv \alpha^{q(x_1+x_2q+\cdots)} \pmod p.$$

Raise both sides to the $(p-1)/q^2$ power to obtain

$$\beta_1^{(p-1)/q^2} \equiv \alpha^{(p-1)(x_1+x_2q+\cdots)/q} \equiv \alpha^{x_1(p-1)/q} (\alpha^{p-1})^{x_2+x_3q+\cdots} \equiv \alpha^{x_1(p-1)/q} \pmod p.$$

The last congruence follows by applying Fermat's theorem. We couldn't

calculate $\beta_1^{(p-1)/q_2}$ as $(\beta_1^p)^{-1/q_2}$ since fractional exponents cause problems. Note that every exponent we have used is an integer.

To find x_1, simply look at the powers

$$\alpha^{k(p-1)/q} \pmod p, \quad k=0, 1, 2, \ldots, q-1,$$

until one of them yields $\beta_1^{(p-1)/q_2}$. Then $x_1=k$.

If $q_3 | p-1$, let $\beta_2 \equiv \beta_1 \alpha^{-x_1 q}$ and raise both sides to the $(p-1)/q_3$ power to obtain x_2. In this way, we can continue until we find that $qr+1$ doesn't divide $p-1$. Since we cannot use fractional exponents, we must stop. But we have determined $x_0, x_1, \ldots, x_{r-1}$, so we know $x \bmod q^r$.

Repeat the procedure for all the prime factors of $p-1$. This yields $x \bmod q_i^{r_i}$ for all i. The Chinese remainder theorem allows us to combine these into a congruence for $x \bmod p-1$. Since $0 \leq x < p-1$, this determines x.

Example

Let $p=41$, $\alpha=7$, and $\beta=12$. We want to solve

$$7x \equiv 12 \pmod{41}.$$

Note that

$$41-1=2^3 \cdot 5.$$

First, let $q=2$ and let's find $x \bmod 2^3$. Write $x \equiv x_0+2x_1+4x_2 \pmod 8$.

To start,

$$\beta^{(p-1)/2} \equiv 1220 \equiv 40 \equiv -1 \pmod{41},$$

and

$$\alpha^{(p-1)/2} \equiv 720 \equiv -1 \pmod{41}.$$

Since

$$\beta^{(p-1)/2} \equiv (\alpha^{(p-1)/2})^{x_0} \pmod{41},$$

we have $x_0=1$. Next,

$$\beta_1 \equiv \beta\alpha^{-x_0} \equiv 12 \cdot 7^{-1} \equiv 31 \pmod{41}.$$

Also,

$$\beta_1^{(p-1)/22} \equiv 31^{10} \equiv 1 \pmod{41}.$$

Since

$$\beta_1^{(p-1)/22} \equiv (\alpha^{(p-1)/2})^{x_1} \pmod{41},$$

we have $x_1=0$. Continuing, we have

$$\beta_2 \equiv \beta_1\alpha^{-2x_1} \equiv 31 \cdot 70 \equiv 31 \pmod{41},$$

and

$\beta^{2(p-1)/q} \equiv 3^{15} \equiv -1 \equiv (\alpha^{(p-1)/2})x_2 \pmod{41}$.

Therefore, $x_2=1$. We have obtained

$x \equiv x_0+2x_1+4x_2 \equiv 1+4 \equiv 5 \pmod 8$.

Now, let $q=5$ and let's find $x \bmod 5$. We have

$\beta^{(p-1)/5} \equiv 12^8 \equiv 18 \pmod{41}$

and

$\alpha^{(p-1)/q} \equiv 7^8 \equiv 37 \pmod{41}$.

Trying the possible values of k yields

$37^0 \equiv 1, 37^1 \equiv 37, 37^2 \equiv 16, 37^3 \equiv 18, 37^4 \equiv 10 \pmod{41}$.

Therefore, 373 gives the desired answer, so $x \equiv 3 \pmod 5$.

Since $x \equiv 5 \pmod 8$ and $x \equiv 3 \pmod 5$, we combine these to obtain
$x \equiv 13 \pmod{40}$, so $x=13$. A quick

calculation checks that
$713 \equiv 12 \pmod{41}$, as desired.

As long as the primes q involved in the preceding algorithm are reasonably small, the calculations can be done quickly. However, when q is large, calculating the numbers $\alpha^{k(p-1)/q}$ for $k=0, 1, 2, \ldots, q-1$ becomes infeasible, so the algorithm no longer is practical. This means that if we want a discrete logarithm to be hard, we should make sure that $p-1$ has a large prime factor.

Note that even if $p-1=tq$ has a large prime factor q, the algorithm can determine discrete logs mod t if t is composed of small prime factors. For this reason, often β is chosen to be a power of α^t. Then the discrete log is

automatically 0 mod t, so the discrete log hides only mod q information, which the algorithm cannot find. If the discrete log x represents a secret (or better, t times a secret), this means that an attacker does not obtain partial information by determining x mod t, since there is no information hidden this way. This idea is used in the Digital Signature Algorithm, which we discuss in Chapter 13.

10.2.2 Baby Step, Giant Step

Eve wants to find x such that $\alpha^x \equiv \beta \pmod{p}$. She does the following. First, she chooses an integer N with $N^2 \geq p-1$, for example $N = \lceil \sqrt{p-1} \rceil$ (where $\lceil x \rceil$ means round x up to the nearest

integer). Then she makes two lists:

1. $\alpha j \pmod p$ for $0 \le j < N$
2. $\beta\alpha^{-Nk} \pmod p$ for $0 \le k < N$

She looks for a match between the two lists. If she finds one, then

$$\alpha^j \equiv \beta\alpha^{-Nk},$$

so $\alpha^{j+Nk} \equiv \beta$. Therefore, $x=j+Nk$ solves the discrete log problem.

Why should there be a match? Since $0 \le x < p-1 \le N^2$, we can write x in base N as $x=x_0+Nx_1$ with $0 \le x_0, x_1 < N$. In fact, $x_1=[x/N]$ and $x_0=x-Nx_1$. Therefore,

$$j=x_0,\ k=x_1$$

gives the desired match.

The list αj for $j=0, 1, 2, \ldots$ is the set of "Baby Steps" since the elements of the list are obtained by multiplying by α, while the "Giant Steps" are obtained in the second list by multiplying by α^{-N}. It is, of course, not necessary to compute all of the second list. Each element, as it is computed, can be compared with the first list. As soon as a match is found, the computation stops.

The number of steps in this algorithm is proportional to $N \approx p$ and it requires storing approximately N numbers. Therefore, the method works for primes p up to 1020, or even slightly larger, but is impractical for very large p.

For an example, see Example 35 in the Computer Appendices.

10.2.3 The Index Calculus

The idea is similar to the method of factoring in Subsection 9.4.1 . Again, we are trying to solve $\beta \equiv \alpha x \pmod{p}$, where p is a large prime and α is a primitive root.

First, there is a precomputation step. Let B be a bound and let p1, p2, …, pm, be the primes less than B. This set of primes is called our **factor base**. Compute $\alpha k \pmod{p}$ for several values of k. For each such number, try to write it as a product of the primes less than B. If this is not the case, discard αk. However, if $\alpha k \equiv \prod piai \pmod{p}$, then

$k \equiv \sum aiL\alpha(pi) \pmod{p-1}$.

When we obtain enough such relations,

we can solve for $L_\alpha(p_i)$ for each i.

Now, for random integers r, compute $\beta\alpha^r \pmod p$. For each such number, try to write it as a product of primes less than B. If we succeed, we have $\beta\alpha^r \equiv \prod p_i^{b_i} \pmod p$, which means

$$L_\alpha(\beta) \equiv -r + \sum b_i L_\alpha(p_i) \pmod{p-1}.$$

This algorithm is effective if p is of moderate size. This means that p should be chosen to have at least 200 digits, maybe more, if the discrete log problem is to be hard.

Example

Let $p=131$ and $\alpha=2$. Let $B=10$, so we are working with the primes 2,3,5,7. A calculation yields the following:

$$21 \equiv 2 \pmod{131} \quad 28 \equiv 53 \pmod{131} \quad 212 \equiv 5 \cdot 7 \pmod{131} \quad 214 \equiv 32 \pmod{131} \quad 234 \equiv 3 \cdot 52 \pmod{131}.$$

Therefore,

$$1 \equiv L2(2) \pmod{130} \quad 8 \equiv 3L2(5) \pmod{130} \quad 12 \equiv L2(5) + L2(7) \pmod{130} \quad 14 \equiv 2L2(3) \pmod{130} \quad 34 \equiv L2(3) + 2L2(5) \pmod{130}.$$

The second congruence yields $L2(5) \equiv 46 \pmod{130}$. Substituting this into the third congruence yields $L2(7) \equiv -34 \equiv 96 \pmod{130}$. The fourth congruence yields only the value of $L2(3) \pmod{65}$ since $\gcd(2, 130) = 1$. This gives two choices for $L2(3) \pmod{130}$. Of course, we could try them and see which works. Or we could use the fifth congruence to obtain $L2(3) \equiv 72 \pmod{130}$. This finishes the precomputation step.

Suppose now that we want to find

L2(37). Trying a few randomly chosen exponents yields
$37 \cdot 243 \equiv 3 \cdot 5 \cdot 7 \pmod{131}$, so

$$L2(37) \equiv -43 + L2(3) + L2(5) + L2(7) \equiv 41 \pmod{130}.$$

Therefore, L2(37)=41.

Of course, once the precomputation has been done, it can be reused for computing several discrete logs for the same prime p.

10.2.4 Computing Discrete Logs Mod 4

When $p \equiv 1 \pmod 4$, the Pohlig-Hellman algorithm computes discrete logs mod 4 quite quickly. What happens when $p \equiv 3 \pmod 4$? The Pohlig-Hellman algorithm won't work, since it

would require us to raise numbers to the $(p-1)/4$ power, which would yield the ambiguity of a fractional exponent. The surprising fact is that if we have an algorithm that quickly computes discrete logs mod 4 for a prime $p \equiv 3 \pmod 4$, then we can use it to compute discrete logs mod p quickly. Therefore, it is unlikely that such an algorithm exists.

There is a philosophical reason that we should not expect such an algorithm. A natural point of view is that the discrete log should be regarded as a number mod $p-1$. Therefore, we should be able to obtain information on the discrete log only modulo the power of 2 that appears in $p-1$. When $p \equiv 3 \pmod 4$, this means that asking questions about discrete logs

mod 4 is somewhat unnatural. The question is possible only because we normalized the discrete log to be an integer between 0 and p−2. For example, $26 \equiv 216 \equiv 9 \pmod{11}$. We defined L2(9) to be 6 in this case; if we had allowed it also to be 16, we would have two values for L2(9), namely 6 and 16, that are not congruent mod 4. Therefore, from this point of view, we shouldn't even be asking about L2(9) mod4.

We need the following lemma, which is similar to the method for computing square roots mod a prime $p \equiv 3 \pmod{4}$ (see Section 3.9).

Lemma

Let $p \equiv 3 \pmod 4$ be prime, let $r \geq 2$, and let y be an integer. Suppose α and γ are two nonzero numbers mod p such that $\gamma \equiv \alpha 2ry \pmod p$. Then

$$\gamma(p+1)/4 \equiv \alpha 2r-1y \pmod p.$$

Proof.

$$\gamma(p+1)/4 \equiv \alpha(p+1)2r-2y \equiv \alpha 2r-1y(\alpha p-1)2r-2y \equiv \alpha 2r-1y \pmod p.$$

The final congruence is because of Fermat's theorem.

Fix the prime $p \equiv 3 \pmod 4$ and let α be a primitive root. Assume we have a machine that, given an input β, gives the output $L\alpha(\beta) \bmod 4$. As we saw previously, it is easy to compute $L\alpha(\beta) \bmod 2$. So the new information supplied by the machine is really only the second bit of the discrete log.

Now assume $\alpha x \equiv \beta \pmod p$. Let $x=x0+2x1+4x2+\cdots+2nxn$ be the binary expansion of x. Using the $L\alpha(\beta) \pmod 4$ machine, we determine x0 and x1. Suppose we have determined $x0, x1, \ldots, xr-1$ with $r \geq 2$. Let

$$\beta r \equiv \beta\alpha-(x0+\cdots+2r-1xr-1) \equiv \alpha 2r(xr+2xr+1+\cdots).$$

Using the lemma r−1 times, we find

$$\beta r((p+1)/4)r-1 \equiv \alpha 2(xr+2xr+1+\cdots)(\text{mod } p).$$

Applying the $L\alpha \pmod 4$ machine to this equation yields the value of xr. Proceeding inductively, we obtain all the values $x0, x1, \ldots, xn$. This determines x, as desired.

It is possible to make this algorithm more efficient. See, for example, [Stinson1, page 175].

In conclusion, if we believe that finding discrete logs for $p \equiv 3 \pmod 4$ is hard, then so is computing such discrete logs mod 4.

10.3 Bit Commitment

Alice claims that she has a method to predict the outcome of football games. She wants to sell her method to Bob. Bob asks her to prove her method works by predicting the results of the games that will be played this weekend. “No way,” says Alice. “Then you will simply make your bets and not pay me. If you want me to prove my system works, why don’t I show you my predictions for last week’s games?” Clearly there is a problem here. We’ll show how to resolve it.

Here’s the setup. Alice wants to send a bit b, which is either 0 or 1, to Bob.

There are two requirements.

1. Bob cannot determine the value of the bit without Alice's help.

2. Alice cannot change the bit once she sends it.

One way is for Alice to put the bit in a box, put her lock on it, and send it to Bob. When Bob wants the value of the bit, Alice removes the lock and Bob opens the box. We want to implement this mathematically in such a way that Alice and Bob do not have to be in the same room when the bit is revealed.

Here is a solution. Alice and Bob agree on a large prime $p \equiv 3 \pmod 4$ and a primitive root α. Alice chooses a random number $x<p-1$ whose second bit x_1 is b. She sends $\beta \equiv \alpha^x \pmod p$ to

Bob. We assume that Bob cannot compute discrete logs for p. As pointed out in the last section, this means that he cannot compute discrete logs mod 4. In particular, he cannot determine the value of b=x1. When Bob wants to know the value of b, Alice sends him the full value of x, and by looking at x mod4, he finds b. Alice cannot send a value of x different than the one already used, since Bob checks that $\beta \equiv \alpha x \pmod{p}$, and this equation has a unique solution $x<p-1$.

Back to football: For each game, Alice sends b=1 if she predicts the home team will win, b=0 if she predicts it will lose. After the game has been played, Alice reveals the bit to Bob, who can see

whether her predictions were correct. In this way, Bob cannot profit from the information by receiving it before the game, and Alice cannot change her predictions once the game has been played.

Bit commitment can also be accomplished with many other one-way functions. For example, Alice can take a random 100-bit string, followed by the bit b, followed by another 100-bit string. She applies the one-way function to this string and sends the result to Bob. After the game, she sends the full 201-bit string to Bob, who applies the one-way function and compares with what Alice originally sent.

10.4 Diffie-Hellman Key Exchange

An important problem in cryptography is how to establish keys for use in cryptographic protocols such as DES or AES, especially when the two parties are widely separated. Public key methods such as RSA provide one solution. In the present section, we describe a different method, due to Diffie and Hellman, whose security is very closely related to the difficulty of computing discrete logarithms.

There are several technical implementation issues related to any key distribution scheme. Some of these are

discussed in Chapter 15. In the present section, we restrict ourselves to the basic Diffie-Hellman algorithm. For more discussion of some security concerns about implementations of the Diffie-Hellman protocol, see [Adrian et al.].

Here is how Alice and Bob establish a private key K. All of their communications in the following algorithm are over public channels.

1. Either Alice or Bob selects a large prime number p for which the discrete logarithm problem is hard and a primitive root α (mod p). Both p and α can be made public.

2. Alice chooses a secret random x with $1 \leq x \leq p-2$, and Bob selects a secret random y with $1 \leq y \leq p-2$.

3. Alice sends α^x (mod p) to Bob, and Bob sends α^y (mod p) to Alice.

4. Using the messages that they each have received, they can each calculate the session key K. Alice calculates K by $K \equiv (\alpha y)x \pmod p$, and Bob calculates K by $K \equiv (\alpha x)y \pmod p$.

There is no reason that Alice and Bob need to use all of K as their key for their communications. Now that they have the same number K, they can use some prearranged procedure to produce a key. For example, they could use the middle 56 bits of K to obtain a DES key.

Suppose Eve listens to all the communications between Alice and Bob. She will know αx and αy. If she can compute discrete logs, then she can find the discrete log of αx to obtain x. Then she raises αy to the power x to obtain $\alpha xy \equiv K$. Once Eve has K, she

can use the same procedure as Alice and Bob to extract a communication key. Therefore, if Eve can compute discrete logs, she can break the system.

However, Eve does not necessarily need to compute x or y to find K. What she needs to do is solve the following:

Computational Diffie-Hellman Problem: Let p be prime and let α be a primitive root mod p. Given $\alpha^x \pmod p$ and $\alpha^y \pmod p$, find $\alpha^{xy} \pmod p$.

It is not known whether or not this problem is easier than computing discrete logs. The reasoning above shows that it is no harder than computing discrete logs. A related problem is the following:

Decision Diffie-Hellman Problem: Let p be prime and let α be a primitive root mod p. Given $\alpha^x \pmod p$ and $\alpha^y \pmod p$, and $c \equiv 0 \pmod p$, decide whether or not $c \equiv \alpha^{xy} \pmod p$.

In other words, if Eve claims that she has found c with $c \equiv \alpha^{xy} \pmod p$, and offers to sell you this information, can you decide whether or not she is telling the truth? Of course, if you can solve the computational Diffie-Hellman problem, then you simply compute $\alpha^{xy} \pmod p$ and check whether it is c (and then you can ignore Eve's offer).

Conversely, does a method for solving the decision Diffie-Hellman problem yield a solution to the computational Diffie-Hellman problem? This is not

known at present. One obvious method is to choose many values of c and check each value until one equals $\alpha^{xy} \pmod{p}$. But this brute force method takes at least as long as computing discrete logarithms by brute force, so is impractical. There are situations involving elliptic curves, analogous to the present setup, where a fast solution is known for the decision Diffie-Hellman problem but no practical solution is known for the computational Diffie-Hellman problem (see Exercise 8 in Chapter 22).

10.5 The ElGamal Public Key Cryptosystem

In Chapter 9, we studied a public key cryptosystem whose security is based on the difficulty of factoring. It is also possible to design a system whose security relies on the difficulty of computing discrete logarithms. This was done by ElGamal in 1985. This system does not quite fit the definition of a public key cryptosystem given at the end of Chapter 9, since the set of possible plaintexts (integers mod p) is not the same as the set of possible ciphertexts (pairs of integers (r, t) mod p). However, this technical point will not concern us.

Alice wants to send a message m to Bob. Bob chooses a large prime p and a primitive root α. Assume m is an integer with $0 \leq m < p$. If m is larger, break it into smaller blocks. Bob also chooses a secret integer b and computes $\beta \equiv \alpha^b \pmod{p}$. The information (p, α, β) is made public and is Bob's public key. Alice does the following:

1. Downloads (p, α, β)
2. Chooses a secret random integer k and computes $r \equiv \alpha^k \pmod{p}$
3. Computes $t \equiv \beta^k m \pmod{p}$
4. Sends the pair (r, t) to Bob

Bob decrypts by computing

$tr^{-b} \equiv m \pmod{p}$.

This works because

$tr-b \equiv \beta km(\alpha k)-b \equiv (\alpha b)km\alpha-bk \equiv m \pmod{p}$.

If Eve determines b, then she can also decrypt by the same procedure that Bob uses. Therefore, it is important for Bob to keep b secret. The numbers α and β are public, and $\beta \equiv \alpha b \pmod{p}$. The difficulty of computing discrete logs is what keeps b secure.

Since k is a random integer, βk is a random nonzero integer mod p. Therefore, $t \equiv \beta km \pmod{p}$ is m multiplied by a random integer, and t is random mod p (unless m=0, which should be avoided, of course). Therefore, t gives Eve no information about m. Knowing r does not seem to give Eve enough additional information.

The integer k is difficult to determine

from r, since this is again a discrete logarithm problem. However, if Eve finds k, she can then calculate $t\beta - k$, which is m.

It is important that a different random k be used for each message. Suppose Alice encrypts messages m1 and m2 for Bob and uses the same value k for each message. Then r will be the same for both messages, so the ciphertexts will be (r, t1) and (r, t2). If Eve finds out the plaintext m1, she can also determine m2, as follows. Note that

$$t1/m1 \equiv \beta k \equiv t2/m2 \pmod{p}.$$

Since Eve knows t1 and t2, she computes $m2 \equiv t2m1/t1 \pmod{p}$.

In Chapter 21, we'll meet an analog of

the ElGamal method that uses elliptic curves.

10.5.1 Security of ElGamal Ciphertexts

Suppose Eve claims to have obtained the plaintext m corresponding to an RSA ciphertext c. It is easy to verify her claim: Compute me (mod n) and check whether this equal c. Now suppose instead that Eve claims to possess the message m corresponding to an ElGamal encryption (r, t). Can you verify her claim? It turns out that this is as hard as the decision Diffie-Hellman problem from Section 10.4. In this aspect, the ElGamal algorithm is therefore much different than the RSA algorithm (of course, if some

randomness is added to an RSA plaintext through OAEP, for example, then RSA encryption has a similar property).

Proposition

A machine that solves Decision Diffie-Hellman problems mod p can be used to decide the validity of mod p ElGamal ciphertexts, and a machine that decides the validity of mod p ElGamal ciphertexts can be used to solve Decision Diffie-Hellman problems mod p.

Proof. Suppose first that you have a machine M1 that can decide whether an ElGamal decryption is correct. In other words, when given the inputs

p, α, β, (r, t), m, the machine outputs "yes" if m is the decryption of (r, t) and outputs "no" otherwise. Let's use this machine to solve the decision Diffie-Hellman problem. Suppose you are given α^x and α^y, and you want to decide whether or not $c \equiv \alpha^{xy}$. Let $\beta = \alpha^x$ and $r = \alpha^y \pmod p$. Moreover, let $t = c$ and $m = 1$. Input

p, α, β, (α^x, α^{xy}), 1

into M1. Note that in the present setup, x is the secret integer b and α^y takes the place of the $r \equiv \alpha^k$. The correct decryption of $(r, t) = (\alpha^y, \alpha^{xy})$ is $tr^{-b} \equiv cr^{-x} \equiv c\alpha^{-xy}$. Therefore, M1 outputs "yes" exactly when $m = 1$ is the same as $c\alpha^{-xy} \pmod p$, namely when $c \equiv \alpha^{xy} \pmod p$. This solves the

decision Diffie-Hellman problem.

Conversely, suppose you have a machine M2 that can solve the decision Diffie-Hellman problem. This means that if you give M2 inputs p, α, αx, αy, c, then M2 outputs “yes” if $c \equiv \alpha^{xy}$ and outputs “no” if not. Let m be the claimed decryption of the ElGamal ciphertext (r, t). Input $\beta \equiv \alpha^{b}$ as α^{x}, so x=b, and input $r \equiv \alpha^{k}$ as α^{y} so y=k. Input $tm^{-1} \pmod p$ as c. Note that m is the correct plaintext for the ciphertext (r, t) if and only if $m \equiv tr^{-a} \equiv t\alpha^{-xy}$, which happens if and only if $tm^{-1} \equiv \alpha^{xy}$. Therefore, m is the correct plaintext if and only if $c \equiv tm^{-1}$ is the solution to the Diffie-Hellman problem. Therefore, with these inputs, M2 outputs

“yes” exactly when m is the correct plaintext.

The reasoning just used can also be used to show that solving the computational Diffie-Hellman problem is equivalent to breaking the ElGamal system:

Proposition

A machine that solves computational Diffie-Hellman problems mod p can be used to decrypt mod p ElGamal ciphertexts, and a machine that decrypts mod p ElGamal ciphertexts can be used to solve computational Diffie-Hellman problems mod p.

Proof. If we have a machine M3 that can decrypt all ElGamal ciphertexts, then

input $\beta \equiv \alpha x$ (so $a=x$) and $r \equiv \alpha y$. Take any nonzero value for t. Then M3 outputs $m \equiv tr-a \equiv t\alpha-xy$. Therefore, $tm \equiv 1 \pmod{p}$ yields the solution αxy to the computational Diffie-Hellman problem.

Conversely, suppose we have a machine M4 that can solve computational Diffie-Hellman problems. If we have an ElGamal ciphertext (r, t), then we input $\alpha x=\alpha a \equiv \beta$ and $\alpha y \equiv \alpha k \equiv r$. Then M4 outputs $\alpha xy \equiv \alpha ak$. Since $m \equiv tr-a \equiv t\alpha-ak$, we obtain the plaintext m.

10.6 Exercises

1.
    1. Let p=13. Compute L2(3).
    2. Show that L2(11)=7.

2.
    1. Let p=17. Compute L3(2).
    2. Show that L3(15)=6.

3.
    1. Compute 65 (mod 11).
    2. Let p=11. Then 2 is a primitive root. Suppose $2x \equiv 6 \pmod{11}$. Without finding the value of x, determine whether x is even or odd.

4. Let p=19. Then 2 is a primitive root. Use the Pohlig-Hellman method to compute L2(14).

5. It can be shown that 5 is a primitive root for the prime 1223. You want to solve the discrete logarithm problem $5x \equiv 3 \pmod{1223}$. Given that

$3611 \equiv 1 \pmod{1223}$, determine whether x is even or odd.

6.

1. Let α be a primitive root mod p. Show that

   $$L_\alpha(\beta_1\beta_2) \equiv L_\alpha(\beta_1)+L_\alpha(\beta_2) \pmod{p-1}.$$

   (Hint: You need the proposition in Section 3.7.)

2. More generally, let α be arbitrary. Show that

   $$L_\alpha(\beta_1\beta_2) \equiv L_\alpha(\beta_1)+L_\alpha(\beta_2) \pmod{\mathrm{ord}_p(\alpha)},$$

   where $\mathrm{ord}_p(\alpha)$ is defined in Exercise 53 in Chapter 3.

7. Let p=101, so 2 is a primitive root. It can be shown that $L_2(3)=69$ and $L_2(5)=24$.

   1. Using the fact that $24=23\cdot 3$, evaluate $L_2(24)$.

   2. Using the fact that $53 \equiv 24 \pmod{101}$, evaluate $L_2(24)$.

8. The number 12347 is prime. Suppose Eve discovers that $210000\cdot 79 \equiv 25431 \pmod{12347}$.

Find an integer k with $0<k<12347$ such that $2k \equiv 79 \pmod{12347}$.

9. Suppose you know that

   $36 \equiv 44 \pmod{137}$, $310 \equiv 2 \pmod{137}$.

   Find a value of x with $0 \le x \le 135$ such that $3x \equiv 11 \pmod{137}$.

10. Let p be a large prime and suppose $\alpha 1018 \equiv 1 \pmod{p}$. Suppose $\beta \equiv \alpha k \pmod{p}$ for some integer k.

    1. Explain why we may assume that $0 \le k<1018$.

    2. Describe a BabyStep, Giant Step method to find k. (Hint: One list can contain numbers of the form $\beta\alpha-109j$.)

11.

    1. Suppose you have a random 500-digit prime p. Suppose some people want to store passwords, written as numbers. If x is the password, then the number $2x \pmod{p}$ is stored in a file. When y is given as a password, the number $2y \pmod{p}$ is compared with the entry for the user in the file. Suppose someone

gains access to the file. Why is it hard to deduce the passwords?

2. Suppose p is instead chosen to be a five-digit prime. Why would the system in part (a) not be secure?

12. Let's reconsider Exercise 55 in Chapter 3 from the point of view of the Pohlig-Hellman algorithm. The only prime q is 2. For k as in that exercise, write $k=x0+2x1+\cdots+215x15$.

    1. Show that the Pohlig-Hellman algorithm yields

       $x0=x1=\cdots=x10=0$

       and

       $2=\beta=\beta1=\cdots=\beta11.$

    2. Use the Pohlig-Hellman algorithm to compute k.

13. In the Diffie-Hellman Key Exchange protocol, suppose the prime is p=17 and the primitive root is $\alpha=3$. Alice's secret is a=3 and Bob's secret is b=5. Describe what Alice and Bob send each other and determine the shared secret that they obtain.

14. In the Diffie-Hellman Key Exchange protocol,

Alice thinks she can trick Eve by choosing her secret to be a=0. How will Eve recognize that Alice made this choice?

15. In the Diffie-Hellman key exchange protocol, Alice and Bob choose a primitive root α for a large prime p. Alice sends $x1 \equiv \alpha a \pmod{p}$ to Bob, and Bob sends $x2 \equiv \alpha b \pmod{p}$ to Alice. Suppose Eve bribes Bob to tell her the values of b and x2. However, he neglects to tell her the value of α. Suppose $\gcd(b, p-1)=1$. Show how Eve can determine α from the knowledge of p, x2, and b.

16. In the ElGamal cryptosystem, Alice and Bob use $p=17$ and $\alpha=3$. Bob chooses his secret to be $a=6$, so $\beta=15$. Alice sends the ciphertext $(r, t)=(7, 6)$. Determine the plaintext m.

17. Consider the following Baby Step, Giant Step attack on RSA, with public modulus n. Eve knows a plaintext m and a ciphertext c with $\gcd(c, n)=1$. She chooses $N2 \geq n$ and makes two lists: The first is $cj \pmod{n}$ for $0 \leq j<N$. The second is $mc^{-Nk} \pmod{n}$ for $0 \leq k<N$.

    1. Why is there always a match between the two lists, and how does a match allow Eve to find the decryption exponent d?

    2. Your answer to (a) is probably partly

false. What you have really found is an exponent d such that $cd \equiv m \pmod{n}$. Give an example of a plaintext–ciphertext pair where the d you find is not the encryption exponent. (However, usually d is very close to being the correct decryption exponent.)

3. Why is this not a useful attack on RSA? (Hint: How long are the lists compared to the time needed to factor n by trial division?)

18. Alice and Bob are using the ElGamal public key cryptosystem, but have set it up so that only Alice, Bob, and a few close associates (not Eve) know Bob's public key. Suppose Alice is sending the message *dismiss Eve* to Bob, but Eve intercepts the message and prevents Bob from receiving it. How can Eve change the message to *promote Eve* before sending it to Bob?

10.7 Computer Problems

1. Let p=53047. Verify that L3(8576)=1234.

2. Let p=31. Evaluate L3(24).

3. Let p=3989. Then 2 is a primitive root mod p.

    1. Show that L2(3925)=2000 and L2(1046)=3000.

    2. Compute L2(3925·1046). (Note: The answer should be less than 3988.)

4. Let p=1201.

    1. Show that $111200/q \equiv 1 \pmod{1201}$ for q=2, 3, 5.

    2. Use method of Exercise 54 in Chapter 3 plus the result of part (a) to show that 11 is a primitive root mod 1201.

    3. Use the Pohlig-Hellman algorithm to find L11(2).

4. Use the Baby Step, Giant Step method to find $L_{11}(2)$.

Chapter 11 Hash Functions

11.1 Hash Functions

A basic component of many cryptographic algorithms is what is known as a hash function. When a hash function satisfies certain non-invertibility properties, it can be used to make many algorithms more efficient. In the following, we discuss the basic properties of hash functions and attacks on them. We also briefly discuss the random oracle model, which is a method of analyzing the security of algorithms that use hash functions. Later, in Chapter 13, hash functions will be used

in digital signature algorithms. They also play a role in security protocols in Chapter 15, and in several other situations.

A **cryptographic hash function** h takes as input a message of arbitrary length and produces as output a **message digest** of fixed length, for example, 256 bits as depicted in Figure 11.1. Certain properties should be satisfied:

1. Given a message m, the message digest h(m) can be calculated very quickly.

2. Given a y, it is computationally infeasible to find an m′ with h(m′)=y (in other words, h is a **one-way**, or **preimage resistant**, function). Note that if y is the message digest of some message, we are not trying to find this message. We are only looking for some m′ with h(m′)=y.

3. It is computationally infeasible to find messages m1 and m2 with h(m1)=h(m2) (in this case, the

function h is said to be **strongly collision resistant**).

Figure 11.1 A Hash Function

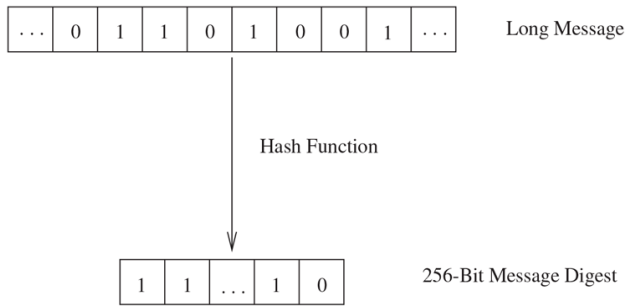


Figure 11.1 Full Alternative Text

Note that since the set of possible messages is much larger than the set of possible message digests, there should always be many examples of messages m1 and m2 with h(m1)=h(m2). The requirement (3) says that it should be hard to find examples. In particular, if Bob produces a message m and its hash h(m), Alice wants to be reasonably

certain that Bob does not know another message m' with $h(m')=h(m)$, even if both m and m' are allowed to be random strings of symbols.

Preimage resistance and collision resistance are closely related, but we list them separately because they are used in slightly different circumstances. The following argument shows that, for our hash functions, collision resistance implies preimage resistance: Suppose H is not preimage resistant. Take a random x and compute $y=H(x)$. If H is not preimage resistant, we can quickly find x' with $H(x')=y=H(x)$. Because H is many-to-one, it is likely that $x \neq x'$, so we have a collision, contradicting the collision resistance of H. However,

there are examples that show that for arbitrary functions, collision resistance does not imply preimage resistance. See Exercise 12.

In practice, it is sometimes sufficient to weaken (3) to require H to be **weakly collision resistant**. This means that given x, it is computationally infeasible to find $x' \neq x$ with $H(x')=H(x)$. This property is also called **second preimage resistance**.

Requirement (3) is the hardest one to satisfy. In fact, in 2004, Wang, Feng, Lai, and Yu (see [Wang et al.]) found many examples of collisions for the popular hash functions MD4, MD5, HAVAL-128, and RIPEMD. The MD5 collisions have been used by Ondrej

Mikle [Mikle] to create two different and meaningful documents with the same hash, and the paper [Lenstra et al.] shows how to produce examples of X.509 certificates (see Section 15.5) with the same MD5 hash (see also Exercise 15). This means that a valid digital signature (see Chapter 13) on one certificate is also valid for the other certificate, hence it is impossible for someone to determine which is the certificate that was legitimately signed by a Certification Authority. It has been reported that weaknesses in MD5 were part of the design of the Flame malware, which attacked several computers in the Middle East, including Iran's oil industry, from 2010 to 2012.

In 2005, Wang, Yin, and Yu [Wang et al. 2] predicted that collisions could be found for the hash function SHA-1 with around 269 calculations, which is much better than the expected 280 calculations required by the birthday attack (see Section 12.1). In addition, they found collisions in a smaller 60-round version of SHA-1. These weaknesses were a cause for concern for using these hash algorithms and led to research into replacements. Finally, in 2017, a joint project between CWI Amsterdam and Google Research found collisions for SHA-1 [Stevens et al.]. Although SHA-1 is still common, it is starting to be used less and less.

One of the main uses of hash functions

is in digital signatures. Since the length of a digital signature is often at least as long as the document being signed, it is much more efficient to sign the hash of a document rather than the full document. This will be discussed in Chapter 13.

Hash functions may also be employed as a check on data integrity. The question of data integrity comes up in basically two scenarios. The first is when the data (encrypted or not) are being transmitted to another person and a noisy communication channel introduces errors to the data. The second occurs when an observer rearranges the transmission in some manner before it gets to the receiver. Either way, the data have become corrupted.

For example, suppose Alice sends Bob long messages about financial transactions with Eve and encrypts them in blocks. Perhaps Eve deduces that the tenth block of each message lists the amount of money that is to be deposited to Eve's account. She could easily substitute the tenth block from one message into another and increase the deposit.

In another situation, Alice might send Bob a message consisting of several blocks of data, but one of the blocks is lost during transmission. Bob might never realize that the block is missing.

Here is how hash functions can be used. Say we send (m, h(m)) over the communications channel and it is

received as (M, H). To check whether errors might have occurred, the recipient computes h(M) and sees whether it equals H. If any errors occurred, it is likely that $h(M) \neq H$, because of the collision-resistance properties of h.

Example

Let n be a large integer. Let $h(m)=m \pmod n$ be regarded as an integer between 0 and $n-1$. This function clearly satisfies (1). However, (2) and (3) fail: Given y, let m=y. Then h(m)=y. So h is not one-way. Similarly, choose any two values m1 and m2 that are congruent mod n. Then h(m1)=h(m2), so h is not strongly collision resistant.

Example

The following example, sometimes called the discrete log hash function, is due to Chaum, van Heijst, and Pfitzmann [Chaum et al.]. It satisfies (2) and (3) but is much too slow to be used in practice. However, it demonstrates the basic idea of a hash function.

First we select a large prime number p such that q=(p−1)/2 is also prime (see Exercise 15 in Chapter 13). We now choose two primitive roots $\alpha 1$ and $\alpha 2$ for p. Since $\alpha 1$ is a primitive root, there exists a such that $\alpha 1^a \equiv \alpha 2 \pmod{p}$. However, we assume that a is not known (finding a, if not given it in advance, involves solving a discrete log problem, which we assume is hard).

The hash function h will map integers

mod q2 to integers mod p. Therefore, the message digest usually contains approximately half as many bits as the message. This is not as drastic a reduction in size as is usually required in practice, but it suffices for our purposes.

Write $m=x0+x1q$ with $0 \le x0, x1 \le q-1$. Then define

$$h(m) \equiv \alpha 1^{x0} \alpha 2^{x1} \pmod{p}.$$

The following shows that the function h is probably strongly collision resistant.

Proposition

If we know messages $m \neq m'$ with $h(m)=h(m')$, then we can determine the discrete logarithm $a=L\alpha 1(\alpha 2)$.

Proof

Write m=x0+x1q and m′=x0'+x1'q. Suppose

$$\alpha 1^{x0}\alpha 2^{x1} \equiv \alpha 1^{x0'}\alpha 2^{x1'} \pmod{p}.$$

Using the fact that $\alpha 2 \equiv \alpha 1^{a} \pmod{p}$, we rewrite this as

$$\alpha 1^{a(x1-x1')-(x0'-x0)} \equiv 1 \pmod{p}.$$

Since α1 is a primitive root mod p, we know that $\alpha 1^{k} \equiv 1 \pmod{p}$ if and only if $k \equiv 0 \pmod{p-1}$. In our case, this means that

$$a(x1-x1') \equiv x0'-x0 \pmod{p-1}.$$

Let d=gcd(x1-x'1, p-1). There are exactly d solutions to the preceding congruence (see Subsection 3.3.1), and they can be found quickly. By the

choice of p, the only factors of p−1 are 1, 2, q, p−1. Since $0 \le x1, x1' \le q-1$, it follows that $-(q-1) \le x1-x1' \le q-1$. Therefore, if $x1-x1' \neq 0$, then it is a nonzero multiple of d of absolute value less than q. This means that $d \neq q, p-1$, so d=1 or 2. Therefore, there are at most two possibilities for a. Calculate $\alpha 1a$ for each possibility; only one of them will yield $\alpha 2$. Therefore, we obtain a, as desired.

On the other hand, if $x1-x1'=0$, then the preceding yields $x0'-x0 \equiv 0 \pmod{p-1}$. Since $-(q-1) \le x0'-x0 \le q-1$, we must have $x0'=x0$. Therefore, $m=m'$, contrary to our assumption.

It is now easy to show that h is preimage resistant. Suppose we have an algorithm

g that starts with a message digest y and quickly finds an m with h(m)=y. In this case, it is easy to find m1 ≠ m2 with h(m1)=h(m2): Choose a random m and compute y=h(m), then compute g(y). Since h maps q2 messages to p−1=2q message digests, there are many messages m′ with h(m′)=h(m). It is therefore not very likely that m′=m. If it is, try another random m. Soon, we should find a collision, that is, messages m1 ≠ m2 with h(m1)=h(m2). The preceding proposition shows that we can then solve a discrete log problem. Therefore, it is unlikely that such an algorithm g exists.

As we mentioned earlier, this hash function is good for illustrative purposes

but is impractical because of its slow nature. Although it can be computed efficiently via repeated squaring, it turns out that even repeated squaring is too slow for practical applications. In applications such as electronic commerce, the extra time required to perform the multiplications in software is prohibitive.

11.2 Simple Hash Examples

There are many families of hash functions. The discrete log hash function that we described in the previous section is too slow to be of practical use. One reason is that it employs modular exponentiation, which makes its computational requirements about the same as RSA or ElGamal. Even though modular exponentiation is fast, it is not fast enough for the massive inputs that are used in some situations. The hash functions described in this section and the next are easily seen to involve only very basic operations on bits and therefore can be carried out much faster than procedures such as modular

exponentiation.

We now describe the basic idea behind many cryptographic hash functions by giving a simple hash function that shares many of the basic properties of hash functions that are used in practice. This hash function is not an industrial-strength hash function and should never be used in any system.

Suppose we start with a message m of arbitrary length L. We may break m into n-bit blocks, where n is much smaller than L. We denote these n-bit blocks by mj, and thus represent m=[m1, m2, ⋯, ml]. Here l=⌈L/n⌉, and the last block ml is padded with zeros to ensure that it has n bits.

We write the jth block mj as a row vector

$$mj=[\ mj1,\ mj2,\ mj3,\ \cdots,\ mjn\ ],$$

where each mji is a bit.

Now, we may stack these row vectors to form an array. Our hash h(m) will have n bits, where we calculate the ith bit as the XOR along the ith column of the matrix, that is $hi=m1i \oplus m2i \oplus \cdots \oplus mli$. We may visualize this as

$$\begin{matrix}[\ m11 & m12 & \cdots & m1n \\ m21 & m22 & \cdots & m2n \\ \vdots & \vdots & \ddots & \vdots \\ ml1 & ml2 & \cdots & mln\ ] \\ \Downarrow & \Downarrow & \Downarrow & \Downarrow \\ \oplus & \oplus & \oplus & \oplus \\ \Downarrow & \Downarrow & \Downarrow & \Downarrow \\ [\ c1 & c2 & \cdots & cn\ ] \end{matrix} = h(m).$$

This hash function is able to take an arbitrary length message and output an n-bit message digest. It is not considered cryptographically secure, though, since it is easy to find two messages that hash

to the same value (Exercise 9).

Practical cryptographic hash functions typically make use of several other bit-level operations in order to make it more difficult to find collisions. Section 11.4 contains many examples of such operations.

One operation that is often used is bit rotation. We define the right rotation operation

Ry(m)

as the result of shifting m to the right by y positions and wrapping the rightmost y bits around, placing them in leftmost y bit locations. Then R−y(m) gives a similar rotation of m by y places to the left.

We may modify our simple hash function above by requiring that block mj is left rotated by j−1, to produce a new block mj′=R−(j−1)(mj). We may now arrange the mj′ in columns and define a new, simple hash function by XORing these columns. Thus, we get

[m11m12⋯m1nm22m23⋯m21m33m34⋯m32⋮⋱⋮mllml, l+1⋯ml, l−1] ⇓ ⇓ ⇓ ⇓ ⊕ ⊕ ⊕ ⊕ ⇓ ⇓ ⇓ ⇓ [c1 c2 ⋯ cn]=h(m).

This new hash function involving rotations mixes the bits in one position with those in another, but it is still easy to find collisions (Exercise 9). Building a cryptographic hash requires considerably more tricks than just rotating. In later sections, we describe hash functions that are used in practice. They use the techniques of the present

section, coupled with many more ways of mixing the bits.

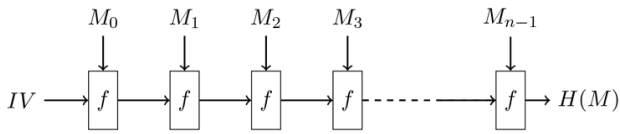
11.3 The Merkle-Damgård Construction

Until recently, most hash functions used a form of the Merkle-Damgård construction. It was invented independently by Ralph Merkle in 1979 and Ivan Damgård in 1989. The main ingredient is a function f, usually called a **compression function**. It takes two bitstrings as inputs, call them H and M, and outputs a bitstring $H'=f(H, M)$ of the same length as H. For example, M could have length 512 and H could have length 256. These are the sizes that the hash function SHA-256 uses, and we'll use them for concreteness. The message that is to be hashed is suitably padded so

that its length is a multiple of 512, and then broken into n blocks of length 512:

M0||M1||M2||···||Mn−1.

An initial value IV is set. Then the blocks are fed one-by-one into f and the final output is the hash value:



11.3-1 Full Alternative Text

This construction is very natural: The blocks are read from the message one at a time and stirred into the mix with the previous blocks. The final result is the hash value.

Over the years, some disadvantages of the method have been discovered. One is called the **length extension attack**.

For example, suppose Alice wants to ensure that her message M to Bob has not been tampered with. They both have a secret key K, so Alice prepends M with K to get K||M. She sends both M and H(K||M) to Bob. Since Bob also knows K, he computes H(K||M) and checks that it agrees with the hash value sent by Alice. If so, Bob concludes that the message is authentic.

Since Eve does not know K, she cannot send her own message M′ along with H(K||M′). But, because of the iterative form of the hash function, Eve can append blocks M″ to M if she intercepts Alice's communication. Then Eve sends

M||M″ and H(K||M||M″)=f(H(K||M), M″)

to Bob. Since she knows H(K||M), she

can produce a message that Bob will regard as authentic. Of course, this attack can be thwarted by using M||K instead of K||M, but it points to a weakness in the construction.

However, using H(M||K) might also cause problems if Eve discovers a way of producing collisions for H. Namely, Eve finds M1 and M2 with H(M1)=H(M2). If Eve can arrange this so that M1 is a good message and M2 is a bad message (see Section 12.1), then Eve arranges for Alice to authenticate M1 by computing H(M1||K), which equals H(M2||K). This means that Alice has also authenticated M2.

Another attack was given by Daum and Luks in 2005. Suppose Alice is using a

high-level document language such as PostScript, which is really a program rather than just a text file. The file begins with a preamble that identifies the file as PostScript and gives some instructions. Then the content of the file follows.

Suppose Eve is able to find random strings R1 and R2 such that

$$H(\text{preamble}; \text{put}(R1)) = H(\text{preamble}; \text{put}(R2)),$$

where put(Ri) instructs the PostScript program to put the string Ri in a certain register. In other words, we are assuming that Eve has found a collision of this form. If any string S is appended to these messages, there is still a collision

H preamble; put(R1)‖S=H preamble; put(R2)‖S

because of the iterative nature of the hash algorithm (we are ignoring the effects of padding).

Of course, Eve has an evil document T1, perhaps saying that Alice (who is a bank president) gives Eve access to the bank's vault. Eve also produces a document T2 that Alice will be willing to sign, for example, a petition to give bank presidents tax breaks. Eve then produces two messages:

Y1=preamble; put(R1);put(R1);if (=)
then T1 else T2Y2=preamble; put(R2);put(R1);if (=)
then T1 else T2.

For example, Y2 puts R1 into a stack, then puts in R2. They are not equal, so T2 is produced. Eve now has two Postscript files, Y1 and Y2, with

H(Y1)=H(Y2). As we'll see in Chapter 13, it is standard for Alice to sign the hash of a message rather than the message itself. Eve shows Y2 to Alice, who compiles it. The output is the petition that Alice is happy to sign. So Alice signs H(Y2). But this means Alice has also signed H(Y1). Eve takes Y1 to the bank, along with Alice's signature on its hash value. The security officer at the bank checks that the signature is valid, then opens the document, which says that Alice grants Eve access to the bank's vault. This potentially costly forgery relies on Eve being able to find a collision, but again it shows a weakness in the construction if there is a possibility of finding collisions.

11.4 SHA-2

In this section and the next, we look at what is involved in making a real cryptographic hash function. Unlike block ciphers, where there are many block ciphers to choose from, there are only a few hash functions that are used in practice. The most notable of these are the Secure Hash Algorithm (SHA) family, the Message Digest (MD) family, and the RIPEMD-160 message digest algorithm. The original MD algorithm was never published, and the first MD algorithm to be published was MD2, followed by MD4 and MD5. Weaknesses in MD2 and MD4 were found, and MD5 was proposed by Ron

Rivest as an improvement upon MD4. Collisions have been found for MD5, and the strength of MD5 is now less certain.

The Secure Hash Algorithm was developed by the National Security Agency (NSA) and given to the National Institute of Standards and Technology (NIST). The original version, often referred to as SHA or SHA-0, was published in 1993 as a Federal Information Processing Standard (FIPS 180). SHA contained a weakness that was later uncovered by the NSA, which led to a revised standards document (FIPS 180-1) that was released in 1995. This revised document describes the improved

version, SHA-1, which for several years was the hash algorithm recommended by NIST. However, weaknesses started to appear and in 2017, a collision was found (see the discussion in Section 11.1). SHA-1 is now being replaced by a series of more secure versions called SHA-2. They still use the Merkle-Damgård construction. In the next section, we'll meet SHA-3, which uses a different construction.

The reader is warned that the discussion that follows is fairly technical and is provided in order to give the flavor of what happens inside a hash function.

The SHA-2 family consists of six algorithms: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and

SHA-512/256. The last three digits indicate the number of bits in the output. We'll describe SHA-256. The other five are very similar.

SHA-256 produces a 256-bit hash and is built upon the same design principles as MD4, MD5, and SHA-1. These hash functions use an iterative procedure. Just as we did earlier, the original message M is broken into a set of fixed-size blocks, $M=M(1)||M(2)||\cdots||M(N)$, where the last block is padded to fill out the block. The message blocks are then processed via a sequence of rounds that use a compression function h' that combines the current block and the result from the previous round. That is, we start with an initial value X0, and

define Xj=h′(Xj−1, M(j)). The final XN is the message digest.

The trick behind building a hash function is to devise a good compression function. This compression function should be built in such a way as to make each input bit affect as many output bits as possible. One main difference between the SHA family and the MD family is that for SHA the input bits are used more often during the course of the hash function than they are for MD4 and MD5. This more conservative approach makes the design of SHA-1 and SHA-2 more secure than either MD4 or MD5, but also makes it a little slower.

In the description of the hash algorithm, we need the following operations on

strings of 32 bits:

1. $X \wedge Y$= bitwise “and”, which is bitwise multiplication mod 2, or bitwise minimum.
2. $X \vee Y$= bitwise “or”, which is bitwise maximum.
3. $X \oplus Y$= bitwise addition mod 2.
4. $\neg X$ changes 1s to 0s and 0s to 1s .
5. X+Y= addition of X and Y mod 232, where X and Y are regarded as integers mod 232.
6. Rn(X)= rotation of X to the right by n positions (the end wraps around to the beginning).
7. Sn(X)= shift of X to the right by n positions, with the first n bits becoming 0s (so the bits at the end disappear and do not wrap around).

We also need the following functions that operate on 32-bit strings:

$Ch(X, Y, Z)=(X \wedge Y) \oplus (\neg X \wedge Z)Maj(X, Y, Z)=(X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z)\Sigma 0($

Define initial hash values

H1(0), H2(0), …, H8(0) as follows:

H1(0)=6A09E667H2(0)=BB67AE85H3(0)=3C6EF372H4(0)=A54FF53AH5(0

The preceding are written in **hexadecimal notation**. Each digit or letter represents a string of four bits:

0=0000, 1=0001, 2=0010, …, 9=1001,
A=1010, B=1011, …, F=1111.

For example, BA1 equals 11*162+10*161+1=2977.

These initial hash values are obtained by using the first eight digits of the fractional parts of the square roots of the first eight primes, expressed as "decimals" in base 16. See Exercise 7.

We also need sixty-four 32-bit words

K0=428A2F98,K1=71374491,… , K63=C67178f2.

They are the first eight hexadecimal digits of the fractional parts of the cube roots of the first 64 primes.

Padding and Preprocessing

SHA-256 begins by taking the original message and padding it with the bit 1 followed by a sequence of 0 bits. Enough 0 bits are appended to make the new message 64 bits short of the next highest multiple of 512 bits in length. Following the appending of 1 and 0s, we append the 64-bit representation of the length T of the message. (This restricts the messages to length less than $264 \approx 1019$ bits, which is not a problem in practice.)

For example, if the original message has

2800 bits, we add a 1 and 207 0s to obtain a new message of length $3008=6\times512-64$. Since $2800=101011110000_2$ in binary, we append fifty-two 0s followed by 101011110000 to obtain a message of length 3072. This is broken into six blocks of length 512.

Break the message with padding into N blocks of length 512:

$$M(1)\|M(2)\|\cdots\|M(N).$$

The hash algorithm inputs these blocks one by one. In the algorithm, each 512-bit block M(i) is divided into sixteen 32-bit blocks:

$$M(i) = M0(i)\|M1(i)\|\cdots\|M15(i).$$

The Algorithm

There are eight 32-bit registers, labeled a, b, c, d, e, f, g, h. These contain the intermediate hash values. The algorithm inputs a block of 512 bits from the message in Step 11, and in Steps 12 through 24, it stirs the bits of this block into a mix with the bits from the current intermediate hash values. After 64 iterations of this stirring, the algorithm produces an output that is added (mod 232) onto the previous intermediate hash values to yield the new hash values. After all of the blocks of the message have been processed, the final intermediate hash values give the hash of the message.

The basic building block of the

algorithm is the set of operations that take place on the subregisters in Steps 15 through 24. They take the subregisters and operate on them using rotations, XORs, and other similar operations.

For more details on hash functions, and for some of the theory involved in their construction, see [Stinson], [Schneier], and [Menezes et al.].

Algorithm 3 The SHA-256 algorithm

- 1: **for** i from 1 to N **do**

  ▷ This initializes the registers with the (i−1)st intermediate hash value

- 2: a←H1(i−1)

- 3: b←H2(i−1)

- 4: c←H3(i−1)
- 5: d←H4(i−1)
- 6: e←H5(i−1)
- 7: f←H6(i−1)
- 8: g←H7(i−1)
- 9: h←H8(i−1)
- 10: **for** k from 0 to 15 **do**
- 11: Wk←Mk(i) ▷ This is where the message blocks are entered.
- 12: **for** j1 from 16 to 63 **do**
- 13: Wj1←σ1(Wj1−2)+Wj1−7+σ0(Wj1−15)+Wj1−16
- 14: **for** j from 0 to 63 **do**
- 15: T1←h+Σ1(e)+Ch(e, f, g)+Kj+Wj
- 16: T2←Σ0(a)+Maj(a, b, c)
- 17: h←g
- 18: g←f

- 19: f←e
- 20: e←d+T1
- 21: d←c
- 22: c←b
- 23: b←a
- 24: a←T1+T2
- 25: H1(i)←a+H1(i−1) ▷ These are the ith intermediate hash values
- 26: H2(i)←b+H2(i−1)
- 27: H3(i)←c+H3(i−1)
- 28: H4(i)←d+H4(i−1)
- 29: H5(i)←e+H5(i−1)
- 30: H6(i)←f+H6(i−1)
- 31: H7(i)←g+H7(i−1)
- 32: H8(i)←h+H8(i−1)
- 33: H(m)=H1(N)||H2(N)||H3(N)||H4(N)||H5(N)||H6(N)||H7(N)||H8(N)

- 34: **return** H(m)

11.5 SHA-3/Keccak

In 2006, NIST announced a competition to produce a new hash function to serve alongside SHA-2. The new function was required to be at least as secure as SHA-2 and to have the same four output possibilities. Fifty-one entries were submitted, and in 2012, **Keccak** was announced as the winner. It was certified as a standard by NIST in 2012 in FIPS-202. (The name is pronounced "ketchak". It has been suggested that the name is related to "Kecak," a type of Balinese dance. Perhaps the movement of the dancers is analogous to the movement of the bits during the algorithm.) This algorithm became the

hash function SHA-3.

The SHA-3 algorithm was developed by Guido Bertoni, Joan Daemen, and Gilles Van Assche from STMicroelectronics and Michaël Peeters from NXP Semiconductors. It differs from the Merkle-Damgård construction and is based on the theory of **Sponge Functions**. The idea is that the first part of the algorithm absorbs the message, and then the hash value is squeezed out. Here is how it works. The **state** of the machine is a string of b bits, which is fed to a function f that takes an input of b bits and outputs a string of b bits, thus producing a new state of the machine. In contrast to the compression functions in the Merkle-Damgård construction, the

function f is a one-to-one function. Such a function could not be used in the Merkle-Damgård situation since the number of input bits (from Mi and the previous step) is greater than the number of output bits. But the different construction in the present case allows it.

Parameters r (“the rate”) and c (“the capacity”) are chosen so that r+c=b. The message (written in binary) is padded so that its length is a multiple of r, then is broken into n blocks of length r:

$$M=M0||M1||M2||\cdots||Mn-1.$$

To start, the state is initialized to all 0s. The absorption stage is the first part of Figure 11.2.

Figure 11.2 A Sponge Function

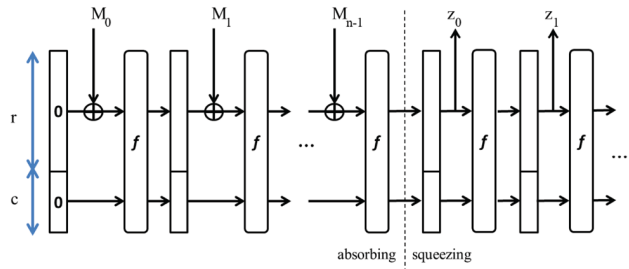


Figure 11.2 Full Alternative Text

After the absorption is finished, the hash value is squeezed out: r bits are output and truncated to the d bits that are used as the hash value. This is the second part of Figure 11.2.

Producing a SHA-3 hash value requires only one squeeze. However, the algorithm can also be used with multiple squeezes to produce arbitrarily long pseudorandom bitstrings. When it is used this way, it is often called SHAKE

(= Secure Hash Algorithm with Keccak).

The “length extension” and collision-based attacks of Section 11.3 are less likely to succeed. Suppose two messages yield the same hash value. This means that when the absorption has finished and the squeezing stage is starting, there are d bits of the state that agree with these bits for the other message. But there are at least c bits that are not output, and there is no reason that these c bits match. If, instead of starting the squeezing, you do another round of absorption, the differing c bits will cause the subsequent states and the outputted hash values to differ. In other words, there are at least 2c possible

internal states for any given d-bit output H.

SHA-3 has four different versions, named SHA3-224, SHA3-256, SHA3-384, and SHA3-512. For SHA3-m, the m denotes the security level, measured in bits. For example, SHA3-256 is expected to require around 2256 operations to find a collision or a preimage. Since 2256 ≈ 1077, this should be impossible well into the future. The parameters are taken to be

b=1600, d=m, c=2m, r=1600−c.

For SHA3-256, these are

b=1600, d=256, c=512, r=1088.

The same function f is used in all versions, which means that it is easy to

change from one security level to another. Note that there is a trade-off between speed and security. If the security parameter m is increased, then r decreases, so the message is read slower, since it is read in blocks of r bits.

In the following, we concentrate on SHA3-256. The other versions are obtained by suitably varying the parameters. For more details, see [FIPS 202].

The Padding. We start with a message M. The message is read in blocks of r=1088 bits, so we want the message to have length that is a multiple of 1088. But first the message is padded to M||01. This is for “domain separation.” There

are other uses of the Keccak algorithm such as SHAKE (mentioned above), and for these other purposes, M is padded differently, for example with `1111`. This initial padding makes it very likely that using M in different situations yields different outputs. Next, "`10*1` padding" is used. This means that first a 1 is appended to M011 to yield M||011 . Then sufficiently many 01 s are appended to make the total length one less than a multiple of 1088. Finally, a 1 is appended. We can now divide the result into blocks of length 1088.

Why are these choices made for the padding? Why not simply append enough `0`s s to get the desired length? Suppose that M1=1010111. Then M1||

10=101011110. Now append 1079 zeros to get the block to be hashed. If M2=1010 is being used in SHAKE, then M2||1111 is padded with 1080 zeros to yield the same block. This means that the outputs for M1 and M2 are equal. The padding is designed to avoid all such situations.

Absorption and Squeezing. From now on, we assume that the padding has been done and we have N blocks of length 1088:

$$M0||M1||\cdots||MN-1.$$

The absorption now proceeds as in Figure 11.2 (we describe the function f later).

1. The initial state S is a string of 0s s of length 1600.

2. For j=0 to N−1, let $S=f(S \oplus (Mj \| 0c))$, where 0c denotes a string of 0s of length c=512. What this does is XOR the message block Mj with the first r=1088 bits of S, and then apply the function f. This yields an updated state S, which is modified during each iteration of the index j.

3. Return S.

The squeezing now proceeds as in Figure 11.2:

1. Input S and let Z be the empty string.

2. While Length(Z)<td (where d=256 is the output size)

    1. Let $Z=Z \| Trunc_r(S)$, where $Trunc_r(S)$ denotes the first r=1088 bits of S.

    2. S=f(S).

3. Return $Trunc_d(Z)$.

The bitstring $Trunc_{256}(Z)$ is the 256-bit hash value SHA-3(M). For the hash

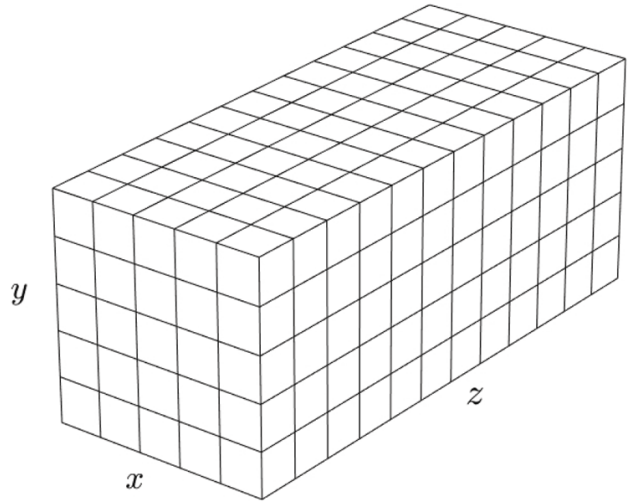
value, we need only one squeeze to obtain Z. But the algorithm could also be used to produce a much longer pseudorandom bitstring, in which case several squeezes might be needed.

The function f. The main component of the algorithm is the function f, which we now describe. The input to f is the 1600-bit state of the machine

$$S=S[0]\|S[1]\|S[2]\|\cdots\|S[1599],$$

where each S[j] is a bit. It's easiest to think of these bits as forming a three-dimensional 5×5×64 array A[x, y, z] with coordinates (x, y, z) satisfying

$$0 \le x \le 4,\ 0 \le y \le 4,\ 0 \le z \le 63.$$



A “column” consists of the five bits with fixed x, z. A “row” consists of the five bits with fixed y, z. A “lane” consists of the 64 bits with fixed x, y.

When we write “for all x, z” we mean for $0 \le x \le 4$ and $0 \le z \le 63$, and similarly for other combinations of x, y, z.

The correspondence between S and A is given by

A[x, y, z]=S[64(5y+x)+z]

for all x, y, z. For example, A[1, 2, 3]=S[707]. The ordering of the indices could be described as "lexicographic" order using y, x, z (not x, y, z), since the index of S corresponding to x1, y1, z1 is smaller than the index for x2, y2, z2 if y1x1z1 precedes y2x2z2 in "alphabetic order."

The coordinates x, y are taken to be numbers mod 5, and z is mod 64. For example A[7, −1, 86] is taken to be A[2, 4, 22], since $7 \equiv 2 \bmod 5$, $-1 \equiv 4 \bmod 5$, and $86 \equiv 22 \bmod 64$.

The computation of f proceeds in several steps. The steps receive the array A as input and they output a modified array A′ to replace A.

The following steps I through V are repeated for i=0 to i=23:

1. The first step XORs the bits in a column with the parities of two nearby columns.
    1. For all x, z, let

       $C[x, z]=A[x, 0, z]\oplus A[x, 1, z]\oplus A[x, 2, z]\oplus A[x, 3, z$

       This gives the “parity” of the bitstring formed by the five bits in the x, z column.

    2. For all x, z, let $D[x, z]=C[x-1, z]\oplus C[x+1, z-1]$.

    3. For all x, y, z, let $A'[x, y, z]=A[x, y, z]\oplus D[x, z]$.

2. The second step rotates the 64 bits in each lane by an amount depending on the lane:
    1. For all z, let $A'[0, 0, z]=A[0, 0, z]$.

    2. Let $(x, y)=(1, 0)$.

    3. For t=0 to 23

1. For all z, let A′[x, y, z]=A[x, y, z−(t+1)(t+2)/2].

2. Let (x, y)=(y, 2x+3y)

4. Return A′.

For example, consider the bits with coordinates of the form (1, 0, z). They are handled by the case t=0 and we have A′[1, 0, z]=A[1, 0, z−1], so the bits in this lane are rotated by one position. Then, in Step 3(b), (x, y)=(1, 0) is changed to (0, 2) for the iteration with t=1. We have A′[0, 2, z]=A[0, 2, z −3], so this lane is rotated by three positions. Then (x, y) is changed to (2, 6), which is reduced mod 5 to (2, 1), and we pass to t=2, which gives a rotation by six (the rotations are by what are known as “triangular numbers”). After t=23, all of the lanes have been rotated.

3. The third step rearranges the positions of the lanes:

1. For all x, y, z, let A′[x, y, z]=A[x +3y, x, z].

Again, the coordinate x+3y should be reduced mod 5.

4. The next step is the only nonlinear step in the algorithm. It XORs each bit with an expression formed from two other bits in its row.

    1. For all x, y, z, let

    2. A′[x, y, z]=A[x, y, z]⊕A[x +1, y, z]⊕1A[x+2, y, z].

    3. The multiplication is multiplying two binary bits, hence is the same as the *AND* operator.

5. Finally, some bits in the (0, 0) lane are modified.

    1. For all x, y, z, let A′[x, y, z]=A[x, y, z].

    2. Set RC=024.

    3. For j=0 to 6, let RC[2j−1]=rc(j+7i), where rc is an auxiliary function defined below.

    4. For all z, let A′[0, 0, z]=A ′[0, 0, z]⊕RC[z].

    5. Return A′.

After I through V are completed for one

value of i, the next value of i is used and I through V are repeated for the new i, through i=23. The final output is the new array A, which yields a new bitstring S of length 1600.

This completes the description of the function f, except that we still need to describe the auxiliary function rc.

The function rc takes an integer t mod 255 as input and outputs a bit according to the following algorithm:

1. If $t \equiv 0 \bmod 255$, return 1 . Else
2. R=10000000
3. For k=1 to t mod 255
    1. Let R=0||R
    2. Let R[0]=R[0]⊕R[8]

3. Let R[4]=R[4]⊕R[8]

4. Let R[5]=R[5]⊕R[8]

5. Let R[6]=R[6]⊕R[8]

6. Let R=Trunc8(R).

4. Return R[0].

The bit that is outputted is rc(t).

11.6 Exercises

1. Let p be a prime and let α be an integer with $p \nmid \alpha$. Let $h(x) \equiv \alpha x \pmod{p}$. Explain why h(x) is not a good cryptographic hash function.

2.
    1. Alice claims that she knows who will win the next World Cup. She takes the name of the team, T, and encrypts it with a one-time pad K, and sends $C=T \oplus K$ to Bob. After the World Cup is finished, Alice reveals K, and Bob computes $T=C \oplus K$ to determine Alice's guess. Why should Bob not believe that Alice actually guessed the correct team, even if $T=C \oplus K$ is correct?

    2. To keep Alice from changing K, Bob requires Alice to send not only $C=T \oplus K$ but also H(K), where H is a good cryptographic hash function. How does the use of the hash function convince Bob that Alice is not changing K?

    3. In the procedure in (b), Bob receives C and H(K). Show how he can determine

Alice's prediction, without needing Alice to send K? (*Hint:* There are fewer than 100 teams T that could win the World Cup.)

3. Let $n=pq$ be the product of two distinct large primes and let $h(x)=x2 \pmod n$.

    1. Why is h preimage resistant? (Of course, there are some values, such as $1, 4, 9, 16, \cdots$ for which it is easy to find a preimage. But usually it is difficult.)

    2. Why is h not strongly collision resistant?

4. Let H(x) be a cryptographic hash function. Nelson tries to make new hash functions.

    1. He takes a large prime p and a primitive root α for p. For an input x, he computes $\beta \equiv \alpha x \pmod p$, then sets $H2(x)=H(\beta)$. The function H2 is not fast enough to be a hash function. Find one other property of hash functions that fails for H2 and one that holds for H2, and justify your answers.

    2. Since his function in part (a) is not fast enough, Nelson tries using $H3=H(x \bmod$

p). This is very fast. Find one other property of hash functions that holds for H3 and one that fails for H3, and justify your answers.

5. Suppose a message m is divided into blocks of length 160 bits: $m=M1\|M2\|\cdots\|M\ell$. Let $h(x)=M1\oplus M2\oplus\cdots\oplus M\ell$. Which of the properties (1), (2), (3) for a hash function does h satisfy and which does it not satisfy? Justify your answers.

6. One way of storing and verifying passwords is the following. A file contains a user's login id plus the hash of the user's password. When the user logs in, the computer checks to see if the hash of the password is the same as the hash stored in the file. The password is not stored in the file. Assume that the hash function is a good cryptographic hash function.

    1. Suppose Eve is able to look at the file. What property of the hash function prevents Eve from finding a password that will be accepted as valid?

    2. When the user logs in, and the hash of the user's password matches the hash stored in the file, what property of the hash function says that the user probably

entered the correct password? (*Hint:* Your answers to (a) and (b) should not be the same.)

7. The initial values Hk(0) in SHA-256 are extracted from the hexadecimal expansions of the fractional parts of the square roots of the first eight primes. Here is what that means.

    1. Compute $\lfloor 232(2-1) \rfloor$ and write the answer in hexadecimal. The answer should be H1(0).

    2. Do a similar computation with 2 replaced by 3, 5, and 19 and compare with the appropriate values of Hk(0).

8. Alice and Bob (and no one else) share a key K. Each time that Alice wants to make sure that she is communicating with Bob, she sends him a random string S of 100 bits. Bob computes B=H(S||K), where H is a good cryptographic hash function, and sends B to Alice. Alice computes H(S||K). If this matches what Bob sent her, she is convinced that she is communicating with Bob.

    1. What property of H convinces Alice that she is communicating with Bob?

2. Suppose Alice's random number generator is broken and she sends the same S each time she communicates with anyone. How can Eve (who doesn't know K, but who intercepts all communications between Alice and Bob) convince Alice that she is Bob?

9. 1. Show that neither of the two hash functions of Section 11.2 is preimage resistant. That is, given an arbitrary y (of the appropriate length), show how to find an input x whose hash is y.

   2. Find a collision for each of the two hash functions of Section 11.2.

10. An unenlightened professor asks his students to memorize the first 1000 digits of π for the exam. To grade the exam, he uses a 100-digit cryptographic hash function H. Instead of carefully reading the students' answers, he hashes each of them individually to obtain binary strings of length 100. Your score on the exam is the number of bits of the hash of your answer that agree with the corresponding bits of the hash of the correct answer.

    1. If someone gets 100% on the exam, why

is the professor confident that the student's answer is correct?

2. Suppose each student gets every digit of π wrong (a very unlikely occurrence!), and they all have different answers. Approximately what should the average on the exam be?

11. A bank in Tokyo is sending a terabyte of data to a bank in New York. There could be transmission errors. Therefore, the bank in Tokyo uses a cryptographic hash function H and computes the hash of the data. This hash value is sent to the bank in New York. The bank in New York computes the hash of the data received. If this matches the hash value sent from Tokyo, the New York bank decides that there was no transmission error. What property of cryptographic hash functions allows the bank to decide this?

12. (Thanks to Danna Doratotaj for suggesting this problem.)

    1. Let H1 map 256-bit strings to 256-bit strings by H1(x)=x. Show that H1 is not preimage resistant but it is collision resistant.

    2. Let H be a good cryptographic hash

function with a 256-bit output. Define a map H2 from binary strings of arbitrary length to binary strings of length 257, as follows. If $0 \leq x < 2256$, let $H2(x)=x\|$ (that is, x with appended). If $x \geq 2256$, let $H2(x)=H(x)\|1$. Show that H2 is collision resistant, and show that if y is a randomly chosen binary string of length 257, then the probability is at least 50% that you can easily find x with $H(x)=y$.

The functions H1 and H2 show that collision resistance does not imply preimage resistance quite as obviously as one might suspect.

13. Show that the computation of rc in Keccak (see the end of Section 11.5) can be given by an LFSR.

14. Let $Maj(X, Y, Z)=(X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z)$ be the function on 32-bit strings in the description of SHA-2.

    1. Suppose that the first bit of X and the first bit of Y are 1 and the first bit of Z is arbitrary. Show that the first bit of $Maj(X, Y, Z)$ is 1.

    2. Suppose that the first bit of X and the first bit of Y are 0 and the first bit of Z is arbitrary. Show that the first bit of

Maj(X, Y, Z) is 0.

This shows that Maj gives the bitwise majority (for 0 vs. 1) for the strings X, Y, Z.

15. Let H be an iterative hash function that operates successively on input blocks of 512 bits. In particular, there is a compression function h and an initial value IV. The hash of a message M1||M2 of 1024 bits is computed by X1=h(IV, M1), and H(M1||M2)=h(X1, M2). Suppose we have found a collision h(IV, x1)=h(IV, x2) for some 512-bit blocks x1 and x2. Choose distinct primes p1 and p2, each of approximately 240 bits. Regard x1 and x2 as numbers between 0 and 2^{512}.

    1. Show that there exists an x0 with $0 \le x0 < p1p2$ such that

       $x0+2^{512}x1 \equiv 0 \pmod{p1}$ and
       $x0+2^{512}x2 \equiv 0 \pmod{p2}$.

    2. Show that if $0 \le k < 230$, then $q1=(x0+2^{512}x1+kp1p2)/p1$ is approximately 2784, and similarly for $q2=(x0+2^{512}x2+kp1p2)/p2$. (Assume that x1 and x2 are approximately 2^{512}.)

    3. Use the Prime Number Theorem (see Section 3.1) to show that the probability

that q1 is prime is approximately 1/543 and the probability that both q1 and q2 are prime is about 1/300000.

4. Show that it is likely that there is some k with $0 \leq k < 230$ such that both q1 and q2 are primes.

5. Show that n1=p1q1 and n2=p2q2 satisfy H(n1)=H(n2).

This method of producing two RSA moduli with the same hash values is based on the method of [Lenstra et al.] for using a collision to produce two X.509 certificates with the same hashes. The method presented here produces moduli n=pq with p and q of significantly different sizes (240 bits and 784 bits), but an adversary does not know this without factoring n.

Chapter 12 Hash Functions: Attacks and Applications

12.1 Birthday Attacks

If there are 23 people in a room, the probability is slightly more than 50% that two of them have the same birthday. If there are 30, the probability is around 70%. This might seem surprising; it is called the **birthday paradox**. Let's see why it's true. We'll ignore leap years (which would slightly lower the probability of a match) and we assume that all birthdays are equally likely (if not, the probability of a match would be slightly higher).

Consider the case of 23 people. We'll compute the probability that they all have different birthdays. Line them up in a row. The first person uses up one day, so the second person has probability (1−1/365) of having a different birthday. There are two days removed for the third person, so the probability is (1−2/365) that the third birthday differs from the first two. Therefore, the probability of all three people having different birthdays is (1−1/365)(1−2/365). Continuing in this way, we see that the probability that all 23 people have different birthdays is

$$(1-\frac{1}{365})(1-\frac{2}{365})\cdots(1-\frac{22}{365})=.493.$$

Therefore, the probability of at least two having the same birthday is

$1-.493=.507$.

One way to understand the preceding calculation intuitively is to consider the case of 40 people. If the first 30 have a match, we're done, so suppose the first 30 have different birthdays. Now we have to choose the last 10 birthdays. Since 30 birthdays are already chosen, we have approximately a 10% chance that a randomly chosen birthday will match one of the first 30. And we are choosing 10 birthdays. Therefore, it shouldn't be too surprising that we get a match. In fact, the probability is 89% that there is a match among 40 people.

More generally, suppose we have N objects, where N is large. There are r people, and each chooses an object

(with replacement, so several people could choose the same one). Then

Prob(there is a match) ≈ 1−e−r2/2N.(12.1)

Note that this is only an approximation that holds for large N; for small n it is better to use the above product and obtain an exact answer. In Exercise 12, we derive this approximation. Choosing r2/2N=ln 2,, we find that if r ≈ 1.177N, then the probability is 50% that at least two people choose the same object.

To summarize, if there are N possibilities and we have a list of length N, then there is a good chance of a match. If we want to increase the chance of a match, we can make the list have length 2N or 5N. The main point is that a length of a constant times N (instead

of something like N) suffices.

For example, suppose we have 40 license plates, each ending in a three-digit number. What is the probability that two of the license plates end in the same three digits? We have N=1000, the number of possible three-digit numbers, and r=40, the number of license plates under consideration. Since

r22N=.8,

the approximate probability of a match is

1−e−.8=.551,

which is more than 50%. We stress that this is only an approximation. The correct answer is obtained by calculating

$1-(1-11000)\ (1-21000)\cdots(1-391000)=.546.$

The next time you are stuck in traffic (and have a passenger to record numbers), check out this prediction.

But what is the probability that one of these 40 license plates has the same last three digits as yours (assuming that yours ends in three digits)? Each plate has probability $1-1/1000$ of not matching yours, so the probability is $(1-1/1000)40=.961$ that none of the 40 plates matches your plate. The reason the birthday paradox works is that we are not just looking for matches between one fixed plate, such as yours, and the other plates. We are looking for matches between any two plates in the set, so there are many more opportunities for

matches.

For more examples, see Examples 36 and 37 in the Computer Appendices.

The applications of these ideas to cryptology require a slightly different setup. Suppose there are two rooms, each with 30 people. What is the probability that someone in the first room has the same birthday as someone in the second room? More generally, suppose there are N objects and there are two groups of r people. Each person from each group selects an object (with replacement). What is the probability that someone from the first group chooses the same object as someone from the second group? In this case,

Prob(there is a match) ≈ 1−e−r2/N.(12.2)

If λ=r2/N, then the probability of exactly i matches is approximately λie −λ/i!. An analysis of this problem, with generalizations, is given in [Girault et al.]. Note that the present situation differs from the earlier problem of finding a match in one set of r people. Here, we have two sets of r people, so a total of 2r people. Therefore, the probability of a match in this set is approximately 1−e−2r2/N. But around half of the time, these matches are between members of the same group, and half the time the matches are the desired ones, namely, between the two groups. The precise effect is to cut the probability down to 1−e−r2/N.

Again, if there are N possibilities and we have two lists of length N, then there is a good chance of a match. Also, if we want to increase the chance of a match, we can make the lists have length 2N or 5N. The main point is that a length of a constant times N (instead of something like N) suffices.

For example, if we take N=365 and r=30, then

$\lambda = 302/365 = 2.466.$

Since $1 - e^{-\lambda} = .915$, there is approximately a 91.5% probability that someone in one group of 30 people has the same birthday as someone in a second group of 30 people.

The birthday attack can be used to find

collisions for hash functions if the output of the hash function is not sufficiently large. Suppose that h is an n-bit hash function. Then there are N=2n possible outputs. Make a list h(x) for approximately r=N=2n/2 random choices of x. Then we have the situation of r ≈ N "people" with N possible "birthdays," so there is a good chance of having two values x1 and x2 with the same hash value. If we make the list longer, for example r=10·2n/2 values of x, the probability becomes very high that there is a match.

Similarly, suppose we have two sets of inputs, S and T. If we compute h(s) for approximately N randomly chosen $s \in S$ and compute h(t) for approximately N

randomly chosen $t \in T$, then we expect some value h(s) to be equal to some value h(t). This situation will arise in an attack on signature schemes in Chapter 13, where S will be a set of good documents and T will be a set of fraudulent documents.

If the output of the hash function is around n=60 bits, the above attacks have a high chance of success. It is necessary to make lists of length approximately $2n/2=230 \approx 109$ and to store them. This is possible on most computers. However, if the hash function outputs 256-bit values, then the lists have length around $2128 \approx 1038$, which is too large, both in time and in memory.

12.1.1 A Birthday Attack on Discrete Logarithms

Suppose we are working with a large prime p and want to evaluate $L_\alpha(h)$. In other words, we want to solve $\alpha x \equiv h \pmod{p}$. We can do this with high probability by a birthday attack.

Make two lists, both of length around p:

1. The first list contains numbers $\alpha k \pmod{p}$ for approximately p randomly chosen values of k.

2. The second list contains numbers $h\alpha^{-\ell} \pmod{p}$ for approximately p randomly chosen values of ℓ.

There is a good chance that there is a match between some element on the first list and some element on the second list. If so, we have

$$\alpha k \equiv h\alpha^{-\ell}, \text{ hence } \alpha k+\ell \equiv h \pmod{p}.$$

Therefore, $x \equiv k+\ell \pmod{p-1}$ is the desired discrete logarithm.

Let's compare this method with the Baby Step, Giant Step (BSGS) method described in Section 10.2. Both methods have running time and storage space proportional to p. However, the BSGS algorithm is **deterministic**, which means that it is guaranteed to produce an answer. The birthday algorithm is **probabilistic**, which means that it probably produces an answer, but this is not guaranteed. Moreover, there is a computational advantage to the BSGS algorithm. Computing one member of a list from a previous one requires one multiplication (by α or by α^{-N}). In the birthday algorithm, the exponent k is

chosen randomly, so αk must be computed each time. This makes the algorithm slower. Therefore, the BSGS algorithm is somewhat superior to the birthday method.

12.2 Multicollisions

In this section, we show that the iterative nature of hash algorithms based on the Merkle-Damgård construction makes them less resistant than expected to finding multicollisions, namely inputs x1, …, xn all with the same hash value. This was pointed out by Joux [Joux], who also gave implications for properties of concatenated hash functions, which we discuss below.

Suppose there are r people and there are N possible birthdays. It can be shown that if $r \approx N(k-1)/k$, then there is a good chance of at least k people having the same birthday. In other words, we

expect a k-collision. If the output of a hash function is random, then we expect that this estimate holds for k-collisions of hash function values. Namely, if a hash function has n-bit outputs, hence N=2n possible values, and if we calculate r=2n(k−1)/k values of the hash function, we expect a k-collision. However, in the following, we'll show that often we can obtain collisions much more easily.

In many hash functions, for example, SHA-256, there is a compression function f that operates on inputs of a fixed length. Also, there is a fixed initial value IV. The message is padded to obtain the desired format, then the following steps are performed:

1. Split the message M into blocks M1, M2, …, Mℓ.

2. Let H0 be the initial value IV.

3. For i=1, 2, …, ℓ, let Hi=f(Hi−1, Mi).

4. Let H(M)=Hℓ.

In SHA-256, the compression function is described in Section 11.4. For each iteration, it takes a 256-bit input from the preceding iteration along with a message block Mi of length 512 and outputs a new string of length 256.

Suppose the output of the function f, and therefore also of the hash function H, has n bits. A birthday attack can find, in approximately 2n/2 steps, two blocks m0 and m0′ such that f(H0, m0)=f(H0, m0′). Let h1=f(H0, m0). A second birthday attack

finds blocks m1 and m1′ with f(h1, m1)=f(h1, m1′). Continuing in this manner, we let

$$h_i = f(h_{i-1}, m_{i-1})$$

and use a birthday attack to find mi and mi′ with

$$f(h_i, m_i) = f(h_i, m_i').$$

This process is continued until we have t pairs of blocks m0, m0′, m1, m1′, …, mt−1, mt−1′, where t is some integer to be determined later.

We claim that each of the 2t messages

$$m_0 \| m_1 \| \cdots \| m_{t-1} m_0' \| m_1 \| \cdots \| m_{t-1} m_0 \| m_1' \| \cdots \| m_{t-1} m_0' \| m_1' \| \cdots \| m_{t-1} \quad \cdots\cdots\cdots m_0' \| m_1 \| \cdots \| m_{t-1}' m_0 \| m_1' \| \cdots \| m_{t-1}' m_0' \| m_1' \| \cdots \| m_{t-1}'$$

(all possible combinations with mi and mi′) has the same hash value. This is

because of the iterative nature of the hash algorithm. At each calculation $h_i=f(m, h_{i-1})$, the same value h_i is obtained whether $m=m_{i-1}$ or $m=m_{i-1}'$. Therefore, the output of the function f during each step of the hash algorithm is independent of whether an m_{i-1} or an m_{i-1}' is used. Therefore, the final output of the hash algorithm is the same for all messages. We thus have a 2t-collision.

This procedure takes approximately $t2^{n/2}$ steps and has an expected running time of approximately a constant times $tn2^{n/2}$ (see Exercise 13). Let $t=2$, for example. Then it takes only around twice as long to find four messages with same hash value as it took to find two

messages with the same hash. If the output of the hash function were truly random, rather than produced, for example, by an iterative algorithm, then the above procedure would not work. The expected time to find four messages with the same hash would then be approximately 23n/4, which is much longer than the time it takes to find two colliding messages. Therefore, it is easier to find multicollisions with an iterative hash algorithm.

An interesting consequence of the preceding discussion relates to attempts to improve hash functions by concatenating their outputs. Suppose we have two hash functions H1 and H2. Before [Joux] appeared, the general

wisdom was that the concatenation

$$H(M)=H1(M)||H2(M)$$

should be a significantly stronger hash function than either H1 or H2 individually. This would allow people to use somewhat weak hash functions to build much stronger ones. However, it now seems that this is not the case. Suppose the output of Hi has ni bits. Also, assume that H1 is calculated by an iterative algorithm, as in the preceding discussion. No assumptions are needed for H2. We may even assume that it is a random oracle, in the sense of Section 12.3. In time approximately 12n2n12n1/2, we can find 2n2/2 messages that all have the same hash value for H1. We then compute the

value of H2 for each of these 2n2/2 messages. By the birthday paradox, we expect to find a match among these values of H2. Since these messages all have the same H1 value, we have a collision for H1||H2. Therefore, in time proportional to 12n2n12n1/2+n22n2/2 (we'll explain this estimate shortly), we expect to be able to find a collision for H1||H2. This is not much longer than the time a birthday attack takes to find a collision for the longer of H1 and H2, and is much faster than the time 2(n1+n2)/2 that a standard birthday attack would take on this concatenated hash function.

How did we get the estimate 12n2n12n1/2+n22n2/2 for the running

time? We used 12n2n12n1/2 steps to get the 2n2/2 messages with the same H1 value. Each of these messages consisted of n2 blocks of a fixed length. We then evaluated H2 for each of these messages. For almost every hash function, the evaluation time is proportional to the length of the input. Therefore, the evaluation time is proportional to n2 for each of the 2n2/2 messages that are given to H2. This gives the term n22n2/2 in the estimated running time.

12.3 The Random Oracle Model

Ideally, a hash function is indistinguishable from a random function. The random oracle model, introduced in 1993 by Bellare and Rogaway [Bellare-Rogaway], gives a convenient method for analyzing the security of cryptographic algorithms that use hash functions by treating hash functions as random oracles.

A random oracle acts as follows. Anyone can give it an input, and it will produce a fixed length output. If the input has already been asked previously by someone, then the oracle outputs the

same value as it did before. If the input is not one that has previously been given to the oracle, then the oracle gives a randomly chosen output. For example, it could flip n fair coins and use the result to produce an n-bit output.

For practical reasons, a random oracle cannot be used in most cryptographic algorithms; however, assuming that a hash function behaves like a random oracle allows us to analyze the security of many cryptosystems that use hash functions.

We already made such an assumption in Section 12.1. When calculating the probability that a birthday attack finds collisions for a hash function, we assumed that the output of the hash

function is randomly and uniformly distributed among all possible outcomes. If this is not the case, so the hash function has some values that tend to occur more frequently than others, then the probability of finding collisions is somewhat higher (for example, consider the extreme case of a really bad hash function that, with high probability, outputs only one value). Therefore, our estimate for the probability of collisions really only applies to an idealized setting. In practice, the use of actual hash functions probably produces very slightly more collisions.

In the following, we show how the random oracle model is used to analyze

the security of a cryptosystem. Because the ciphertext is much longer than the plaintext, the system we describe is not as efficient as methods such as OAEP (see Section 9.2). However, the present system is a good illustration of the use of the random oracle model.

Let f be a one-way one-to-one function that Bob knows how to invert. For example, $f(x)=x^e \pmod n$, where (e, n) is Bob's public RSA key. Let H be a hash function. To encrypt a message m, which is assumed to have the same bitlength as the output of H, Alice chooses a random integer r mod n and lets the ciphertext be

$$(y1, y2)=(f(r), H(r)\oplus m).$$

When Bob receives (y1, y2), he

computes

$$r=f-1(y1), \qquad m=H(r)\oplus y2.$$

It is easy to see that this decryption produces the original message m.

Let's assume that the hash function is a random oracle. We'll show that if Alice can succeed with significantly better than 50% probability, then she can invert f with significantly better than zero probability. Therefore, if f is truly a one-way function, the cryptosystem has the ciphertext indistinguishability property. To test this property, Alice and Carla play the CI Game from Section 4.5. Carla chooses two ciphertexts, m0 and m1, and gives them to Alice. Alice randomly chooses b=0 or 1 and encrypts mb, yielding the ciphertext (y1, y2). She

gives (y1, y2) to Carla, who tries to guess whether b=0 or b=1. Suppose that

Prob(Carla guesses correctly) ≥ 12+ϵ.

Let L={r1, r2, …, rℓ} be the set of y for which Carla can compute f−1(y). If y1 ∈ L, then Carla computes the value r such that f(r)=y1. She then asks the random oracle for the value H(r), computes y2 ⊕ H(r), and obtains mb. Therefore, when y1 ∈ L, Carla guesses correctly.

If r ∉ L, then Carla does not know the value of H(r). Since H is a random oracle, the possible values of H(r) are randomly and uniformly distributed among all possible outputs, so H(m) ⊕ mb is the same as encrypting mb with a one-time pad. As we saw in

Section 4.4, this means that y2 gives Alice no information about whether it comes from m1 or from m2. So if $r \notin L$, Alice has probability 1/2 of guessing the correct plaintext.

Therefore

$$12+\epsilon=\text{Prob(Alice guesses correctly)}=12\text{Prob}(r \notin L)+1\cdot\text{Prob}(r \in L)=121-\text{Prob}(x \in L)+1\cdot\text{Prob}(r \in L)=12+1$$

It follows that $\text{Prob}(x \in L)=2\epsilon$.

If we assume that it is computationally feasible for Alice to find b with probability at most ϵ, then we conclude that it is computationally feasible for Alice to guess r correctly with probability $12+\epsilon$. Therefore, if the function f is one-way, then the cryptosystem has the ciphertext

indistinguishability property.

Note that it was important in the argument to assume that the values of H are randomly and uniformly distributed. If this were not the case, so the hash function had some bias, then Alice might have some method for guessing correctly with better than 50% probability when $r \notin L$. Therefore, the assumption that the hash function is a random oracle is important.

Of course, a good hash function is probably close to acting like a random oracle. In this case, the above argument shows that the cryptosystem with an actual hash function should be fairly resistant to Alice guessing correctly. However, it should be noted that

Canetti, Goldreich, and Halevi [Canetti et al.] have constructed a cryptosystem that is secure in the random oracle model but is not secure for any concrete choice of hash function. Fortunately, this construction is not one that would be used in practice.

The above procedure of reducing the security of a system to the solvability of some fundamental problem, such as the non-invertibility of a one-way function, is common in proofs of security. For example, in Section 10.5, we reduced certain questions for the ElGamal public key cryptosystem to the solvability of Diffie-Hellman problems.

Section 12.2 shows that most hash functions do not behave as random

oracles with respect to multicollisions. This indicates that some care is needed when applying the random oracle model.

The use of the random oracle model in analyzing a cryptosystem is somewhat controversial. However, many people feel that it gives some indication of the strength of the system. If a system is not secure in the random oracle model, then it surely is not safe in practice. The controversy arises when a system is proved secure in the random oracle model. What does this say about the security of actual implementations? Different cryptographers will give different answers. However, at present, there seems to be no better method of

analyzing the security that works widely.

12.4 Using Hash Functions to Encrypt

Cryptographic hash functions are some of the most widely used cryptographic tools, perhaps second only to block ciphers. They find applications in many different areas of information security. Later, in Chapter 13, we shall see an application of hash functions to digital signatures, where the fact that they shrink the representation of data makes the operation of creating a digital signature more efficient. We now look at how they may be used to serve the role of a cipher by providing data confidentiality.

A cryptographic hash function takes an input of arbitrary length and provides a fixed-size output that appears random. In particular, if we have two distinct inputs, then their hashes should be different. Generally, their hashes are very different. This is a property that hash functions share with good ciphers and is a property that allows us to use a hash function to perform encryption.

Using a hash function to perform encryption is very similar to a stream cipher in which the output of a pseudorandom number generator is XORed with the plaintext. We saw such an example when we studied the output feedback mode (OFB) of a block cipher. Much like the block cipher did for OFB,

the hash function creates a pseudorandom bit stream that is XORed with the plaintext to create a ciphertext.

In order to make a cryptographic hash function operate as a stream cipher, we need two components: a key shared between Alice and Bob, and an initialization vector. We shall soon address the issue of the initialization vector, but for now let us begin by assuming that Alice and Bob have established a shared secret key KAB.

Now, Alice could create a pseudorandom byte x1 by taking the leftmost byte of the hash of KAB; that is, x1=L8h(KAB). She could then encrypt a byte of plaintext p1 by XORing with the random byte x1 to

produce a byte of ciphertext

$c1=p1 \oplus x1.$

But if she has more than one byte of plaintext, then how should continue? We use feedback, much like we did in OFB mode. The next pseudorandom byte should be created by $x2=L8h(KAB\|x1)$. Then the next ciphertext byte can be created by

$c2=p2 \oplus x2.$

In general, the pseudorandom byte xj is created by $xj=L8h(KAB\|xj-1)$, and encryption is simply XORing xj with the plaintext pj. Decryption is a simple matter, as Bob must merely recreate the bytes xj and XOR with the ciphertext cj to get out the plaintext pj.

There is a simple problem with this procedure for encryption and decryption. What if Alice wants to encrypt a message on Monday, and a different message on Wednesday? How should she create the pseudorandom bytes? If she starts all over, then the pseudorandom sequence xj on Monday and Wednesday will be the same. This is not desirable.

Instead, we must introduce some randomness to make certain the two bit streams are different. Thus, each time Alice sends a message, she should choose a random initialization vector, which we denote by x0. She then starts by creating x1=L8h(KAB||x0) and proceeding as before. But now she must

send x0 to Bob, which she can do when she sends c1. If Eve intercepts x0, she is still not able to compute x1 since she doesn't know KAB. In fact, if h is a good hash function, then x0 should give no information about x1.

The idea of using a hash function to create an encryption procedure can be modified to create an encryption procedure that incorporates the plaintext, much in the same way as the CFB mode does.

12.5 Message Authentication Codes

When Alice sends a message to Bob, two important considerations are

1. Is the message really from Alice?
2. Has the message been changed during transmission?

Message authentication codes (MAC) solve these problems. Just as is done with digital signatures, Alice creates an appendix, the MAC, that is attached to the message. Thus, Alice sends (m, MAC) to Bob.

12.5.1 HMAC

One of the most commonly used is HMAC (= Hashed MAC), which was invented in 1996 by Mihir Bellare, Ran Canetti, and Hugo Krawczyk and is used in the IPSec and TLS protocols for secure authenticated communications.

To set up the protocol, we need a hash function H. For concreteness, assume that H processes messages in blocks of 512 bits. Alice and Bob need to share a secret key K. If K is shorter than 512 bits, append enough 0s to make its length be 512. If K is longer than 512 bits, it can be hashed to obtain a shorter key, so we assume that K has 512 bits.

We also form innerpadding and outerpadding strings:

```
opad = 5C5C5C … 5C, ipad = 363636 …
36,
```

which are binary strings of length 512, expressed in hexadecimal (`5=0101`, `C=1100`, etc.).

Let m be the message. Then Alice computes

$$\text{HMAC}(m, K) = H((K \oplus \text{opad}) \| H((K \oplus \text{ipad}) \| m)).$$

In other words, Alice first does the natural step of computing $H((K \oplus \text{ipad}) \| m)$. But, as pointed out in Section 11.3, this is susceptible to length extension attacks for hash functions based on the Merkle-Damgård construction. So she prepends $K \oplus \text{opad}$ and hashes again. This seems to be resistant to all known attacks.

Alice now sends the message m, either encrypted or not, along with HMAC(m, K), to Bob. Since Bob knows K, he can compute HMAC(m, K). If it agrees with the value that Alice sent, he assumes that the message is from Alice and that it is the message that Alice sent.

If Eve tries to change m to m′, then $H(K \oplus ipad || m')$ should differ from $H(K \oplus ipad || m)$ (collision resistance) and therefore HMAC(m′, K) should differ from HMAC(K, m) (again, by collision resistance). Therefore, HMAC(m, K) tells Bob that the message is authentic. Also, it seems unlikely that Eve can authenticate a message m without knowing K, so Bob is sure that Alice

sent the message.

For an analysis of the security of HMAC, see [Bellare et al.]

12.5.2 CBC-MAC

An alternative to using hash functions is to use a block cipher in CBC mode. Let EK() be a block cipher, such as AES, using a secret key K that is shared between Alice and Bob. We may create an appendix very similar to the output of a keyed hash function by applying EK() in CBC mode to the entire message m and using the final output of the CBC operation as the MAC.

Suppose that our message m is of the form m=(m1, m2, ⋯, ml), where each

mj is a block of the message that has the same length as the block length of the encryption algorithm EK(). The last block ml may require padding in order to guarantee that it is a full block. Recall that the CBC encryption procedure is given by

$$C_j = EK(m_j \oplus C_{j-1}) \qquad j = 1, 2, \cdots, l,$$

where C0=IV is the initialization vector. The CBC-MAC is then given as

$$MAC = (IV, C_l)$$

and Alice then sends Bob (m, MAC).

CBC-MAC has many of the same concerns as keyed hash functions. For example, CBC-MAC also suffers from its own form of length extension attacks. Let MAC(a) correspond to the CBC-

MAC of a message a. Suppose that Eve has found two messages a and b with MAC(a)=MAC(b). Then Eve knows that MAC(a, x)=MAC(b, x) for an additional block x. If Eve can convince Alice to authenticate a ′=[a, x], she can swap a ′ with b ′=[b, x] to create a forged document that appears valid. Of course, the trick is in convincing Alice to authenticate a ′, but it nonetheless is a concern with CBC-MAC.

When using CBC-MAC, one should also be careful not to use the key K for purposes other than authentication. A different key must be used for message confidentiality than for calculating the message authentication code. In particular, if one uses the same key for

confidentiality as for authentication, then the output of CBC blocks during the encryption of a message for confidentiality could be the basis for a forgery attack against CBC-MAC.

12.6 Password Protocols

We now look at how the communications take place in a password-based authentication protocol, as is commonly used to log onto computer systems. Password-based authentication is a popular approach to authentication because it is much easier for people to remember a phrase (the *password*) than it is to remember a long, cryptographic response.

In a password protocol, we have a user, Alice, and a verifier, Veronica. Veronica is often called a host and might, for example, be a computer terminal or a smartphone or an email service. Alice

and Veronica share knowledge of the password, which is a long-term secret that typically belongs to a much smaller space of possibilities than cryptographic keys and thus passwords have small entropy (that is, much less randomness is in a password).

Veronica keeps a list of all users and their passwords {(ID, PID)}. For many hosts this list might be small since only a few users might log into a particular machine, while in other cases the password list can be more substantial. For example, email services must maintain a very large list of users and their passwords. When Alice wants to log onto Veronica's service, she must contact Veronica and tell her who she is

and give her password. Veronica, in turn, will check to see if Alice's password is legitimate.

A basic password protocol proceeds in several steps:

1. Alice → Veronica: "Hello, I am Alice"
2. Veronica → Alice: "Password?"
3. Alice → Veronica: PAlice
4. Veronica: Examines password file {(ID, PID)} and verifies that the pair (Alice, PAlice) belongs to the password file. Service is granted if the password is confirmed.

This protocol is very straightforward and, as it stands, has many flaws that should stand out. Perhaps one of the most obvious problems is that there is no mutual authentication between Alice

and Veronica; that is, Alice does not know that she is actually communicating with the legitimate Veronica and, vice versa, Veronica does not actually have any proof that she is talking with Alice. While the purpose of the password exchange is to authenticate Alice, in truth there is no protection from message replay attacks, and thus anyone can pretend to be either Alice or Veronica.

Another glaring problem is the lack of confidentiality in the protocol. Any eavesdropper who witnesses the communication exchange learns Alice's password and thus can imitate Alice at a later time.

Lastly, another more subtle concern is

the storage of the password file. The storage of {(ID, PID)} is a design liability as there are no guarantees that a system administrator will not read the password file and leak passwords. Similarly, there is no guarantee that Veronica's system will not be hacked and the password file stolen.

We would like the passwords to be protected while they are stored in the password file. Needham proposed that, instead of storing {(ID, PID)}, Veronica should store {(ID, h(PID))}, where h is a one-way function that is difficult to invert, such as a cryptographic hash function. In this case, Step 4 involves Veronica checking h(PAlice) against {(ID, h(PID))}. This basic scheme is

what was used in the original Unix password system, where the function h was the variant of DES that used Salt (see Chapter 7). Now, an adversary who gets the file {(ID, h(PID))} can't use this to respond in Step 3.

Nevertheless, although the revised protocol protects the password file, it does not address the eavesdropping concern. While we could attempt to address eavesdropping using encryption, such an approach would require an additional secret to be shared between Alice and Veronica (namely an encryption key). In the following, we present two solutions that do not require additional secret information to be shared between Alice and Veronica.

12.6.1 The Secure Remote Password protocol

Alice wants to log in to a server using her password. A common way of doing this is the Secure Remote Password protocol (SRP).

First, the system needs to be set up to recognize Alice and her password. Alice has her login name I and password P. Also, the server has chosen a large prime p such that $(p-1)/2$ is also prime (this is to make the discrete logarithm problem mod p hard) and a primitive root α mod p. Finally, a cryptographic hash function H such as SHA256 or SHA3 is specified.

A random bitstring s is chosen (this is

called “salt”), and $x=H(s||I||P)$ and $v \equiv \alpha x \pmod p$ are computed. The server discards P and x, but stores s and v along with Alice’s identification I. Alice saves only I and P.

When Alice wants to log in, she and the server perform the following steps:

1. Alice sends I to the server and the server retrieves the corresponding s and v.
2. The server sends s to Alice, who computes $x=H(s||I||P)$.
3. Alice chooses a random integer a mod $p-1$ and computes $A \equiv \alpha a \pmod p$. She sends A to the server.
4. The server chooses a random b mod $p-1$, computes $B \equiv 3v+\alpha b \bmod p$, and sends B to Alice.
5. Both Alice and the server compute $u=H(A||B)$.
6. Alice computes $S \equiv (B-3\alpha x)a+ux \pmod{p-1}$ and the server computes S as $(Avu)b$ (these yield the

same S; see below).

7. Alice computes M1=H(A||B||S) and sends M1 to the server, which checks that this agrees with the value of M1 that is computed using the server's values of A, B, S.

8. The server computes M2=H(A||M1||S) and sends M2 to Alice. She checks that this agrees with the value of M2 she computes with her values of A, M1, S.

9. Both Alice and the server compute K=H(S) and use this as the session key for communications.

Several comments are in order. We'll number them corresponding to the steps in the protocol.

1. The server does not directly store P or a hash of P. The hash of P is stored in a form protected by a discrete logarithm problem. Originally, x=H(s||P) was used and the salt s was included to slow down brute force attacks on passwords. Eve can try various values of P, trying to match a value of v, until someone's password is found (this is called a "dictionary attack"). If a sufficiently long bitstring s is included, this attack becomes infeasible. The

current version of SRP includes I in the hash to get x, so Eve needs to attack an individual entry. Since Eve knows an individual's salt (if she obtained access to the password file), the salt is essentially part of the identification and does not slow down the attack on the individual's password.

2. Sending s to Alice means that Alice does not need to remember s.

3. This is essentially the start of a protocol similar to the Diffie-Hellman key exchange.

4. Earlier versions of SRP used $B \equiv v+\alpha b$. But this meant that an attacker posing as the server could choose a random x', compute $v' \equiv \alpha x'$, and use $B \equiv \alpha x'+\alpha b$, thus allowing the attacker to check whether one of b, x' is the hash value x. In effect, this could speed up the attack by a factor of 2. The 3 is included to avoid this.

5. In an earlier version of SRP, u was chosen randomly by the server and sent to Alice. However, if the server sends u before A is received (for example, it might seem more efficient for the server to send both s and u in Step 2), there is a possible attack. See Exercise 16. The present method of having $u=H(A\|B)$ ensures that u is determined after A.

6. Let's show that the two values of S are equal:

   $(B-3\alpha x)a+ux \equiv (B-3v)a+ux \equiv (\alpha b)a+ux \equiv \alpha ab\alpha uxb$

   and

   $(Avu)b \equiv \alpha a(\alpha x)ub \equiv \alpha ab\alpha uxb.$

   Therefore, they agree. Note the hints of the Diffie-Hellman protocol, where αab is computed in two ways.

   Since the value of B changes for each login, the value of S also changes, so an attacker cannot simply reuse some successful S.

7. Up to this point, the server has no assurance that the communications are with Alice. Anyone could have sent Alice's I and sent a random A. Alice and the server have computed S, but they don't know that their values agree. If they do, it is very likely that the correct x, hence the correct P, is being used. Checking M1 shows that the values of S agree because of the collision resistance of the hash function. Of course, if the values of S don't agree, then Alice and the server will produce different session keys K in the last step, so communications will fail for this reason, too. But it seems better to terminate the protocol earlier if something is wrong.

8. How does Alice know that she is communicating with the server? This step tells Alice that the server's value of S matches hers, so it is very likely that the entity she is communicating with knows the correct x. Of course, someone who has hacked into the password file has all the information that the server has and can therefore masquerade as the server. But otherwise Alice is confident that she is communicating with the server.

9. At the point, Alice and the server are authenticated to each other. The session key serves as the secret key for communications between Alice and the server during the current session.

Observe that B, M1, and M2 are the only numbers that are transmitted that depend on the password. The value of B contains $v \equiv \alpha x$, but this is masked by adding on the random number αb. The values of M1 and M2 contain S, which depends on x, but it is safely hidden inside a hash function. Therefore, if is very unlikely that someone who

eavesdrops on communications between Alice and the server will obtain any useful information. For more on the security and design considerations, see [Wu1], [Wu2].

12.6.2 Lamport's protocol

Another method was proposed by Lamport. The protocol, which we now introduce, is an example of what is known as a *one-time* password scheme since each run of the protocol uses a temporary password that can only be used once. Lamport's one-time password protocol is a good example of a special construction using one-way (specifically, hash) functions that shows up in many different applications.

To start, we assume that Alice has a password PAlice, that Veronica has chosen a large integer n, and that Alice and Veronica have agreed upon a one-way function h. A good choice for such an h is a cryptographic hash function, such as SHA256 or SHA3, which we described in Chapter 11. Veronica calculates $h^n(PAlice) = h(h(\cdots(h(PAlice))\cdots))$, and stores Alice's entry (Alice, $h^n(PAlice)$, n) in a password file. Now, when Alice wants to authenticate herself to Veronica, she uses the revised protocol:

1. Alice → Veronica: "Hello, I am Alice."
2. Veronica → Alice: n, "Password?"
3. Alice → Veronica: $r = h^{n-1}(PAlice)$
4. Veronica takes r, and checks to see whether

h(r)=hn(PAlice). If the check passes, then Veronica updates Alice's entry in the password as (Alice, hn−1(PAlice), n−1), and Veronica grants Alice access to her services.

At first glance, this protocol might seem confusing, and to understand how it works in practice it is useful to write out the following chain of hashes, known as a **hash chain**:

hn(PAlice)←hhn−1(PAlice)←hhn−2(PAlice)←h⋯
←hh(PAlice)←hPAlice.

For the first run of the protocol, in step 2, Veronica will tell Alice n and ask Alice the corresponding password that will hash to hn(PAlice). In order for Alice to correctly respond, she must calculate hn−1(PAlice), which she can do since she has the original password. After Alice is successfully verified,

Veronica will throw away hn(PAlice) and update the password file to contain (Alice, hn−1(PAlice), n−1).

Now suppose that Eve saw hn −1(PAlice). This won't help her because the next time the protocol is run, in step 2 Veronica will issue n−1 and thereby ask Alice for the corresponding password that will hash to hn −1(PAlice). Although Eve has hn −1(PAlice), she cannot determine the required response r=hn−2(PAlice).

The protocol continues to run, with Veronica updating her password file until she runs to the end of the hash chain. At that point, Alice and Veronica must renew the password file by changing the initial password PAlice to

a new password.

This protocol that we have just examined is the basis for the S/Key protocol, which was implemented in Unix operating systems in the 1980s. Although the S/Key protocol's use of one-time passwords achieves its purpose of protecting the password exchange from eavesdropping, it is nevertheless weak when one considers an active adversary. In particular, the fact that the counter n in step 2 is sent in the clear, and the lack of authentication, is the basis for an intruder-in-the-middle attack.

In the intruder-in-the-middle attack described below, Alice intends to communicate with Veronica, but the

active adversary Malice intercepts the communications between Alice and Veronica and sends her own.

1. Alice → Malice (Veronica): “Hello, I am Alice.”
2. Malice → Veronica: “Hello, I am Alice.”
3. Veronica → Malice: n, “Password?”
4. Malice → Alice: n−1, “Password?”
5. Alice → Malice: r1=hn−2(PAlice)
6. Malice → Veronica: r2=hn−1(PAlice)=h(hn −2(PAlice)).
7. Veronica takes r2, and checks to see whether h(r2)=hn(PAlice). The check will pass, and Veronica will think she is communicating with Alice, when really she is corresponding with Malice.

One of the problems with the protocol is that there is no authentication of the origin of the messages. Veronica does

not know whether she is really communicating with Alice, and likewise Alice does not have a strong guarantee that she is communicating with Veronica.

The lack of origin authentication also provides the means to launch another clever attack, known as the small value attack. In this attack, Malice impersonates Veronica and asks Alice to respond to a small n. Then Malice intercepts Alice's answer and uses that to calculate the rest of the hash chain. For example, if Malice sent n=10, she would be able to calculate h10(PAlice), h11(PAlice), h12(PAlice), and so on. The small value of n allows Malice to hijack the majority of the hash chain,

and thereby imitate Alice at a later time.

As a final comment, we note that the protocol we described is actually different than what was originally presented by Lamport. In Lamport's original one-time password protocol, he required that Alice and Veronica keep track of n on their own without exchanging n. This has the benefit of protecting the protocol from active attacks by Malice where she attempts to use n to her advantage. Unfortunately, Lamport's scheme required that Alice and Veronica stayed synchronized with each other, which in practice turns out to be difficult to ensure.

12.7 Blockchains

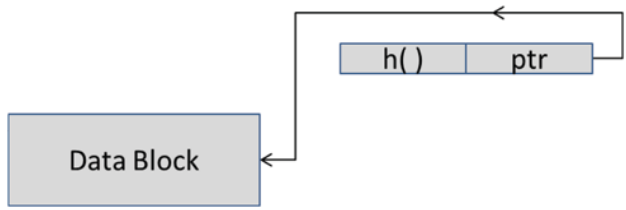
One variation of the concept of a hash chain is the blockchain. Blockchains are a technology that has garnered a lot of attention since they provide a convenient way to keep track of information in a secure and distributed manner.

Hash chains are iterations of hash functions, while blockchains are hash chains that have extra structure. In order to understand blockchains, we need to look at several different building blocks.

Let us start with hash pointers. A **hash pointer** is simply a pointer to where

some data is stored, combined with a cryptographic hash of the value of the data that is being pointed at. We may visualize a hash pointer as something like Figure 12.1.

Figure 12.1 A hash pointer consists of a pointer that references a block of data, and a cryptographic hash of that data



The hash pointer is useful in that it gives a means to detect alterations of the block of data. If someone alters the block, then the hash contained in the hash pointer will not match the hash of

the altered data block, assuming of course that no one has altered the hash pointer.

If we want to make it harder for someone to alter the hash, then we can create an ordered collection of data blocks, each with a hash pointer to a previous block. This is precisely the idea behind a blockchain. A blockchain is a collection of data blocks and hash pointers arranged in a data structure known as a linked list. Normal linked lists involve series of blocks of data that are each paired with a pointer to a previous block of data and its pointer. Blockchains, however, replace the pointer in a linked list with a hash pointer, as depicted in Figure 12.2.

Figure 12.2 A blockchain consists of a collection of blocks. Each block contains a data block, a hash of the previous block, and a pointer to the previous block. A final hash pointer references the end of the blockchain

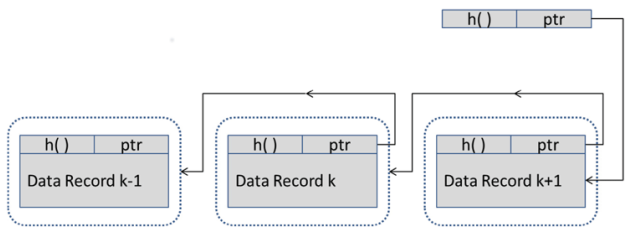


Figure 12.2 Full Alternative Text

Blockchains are useful because they allow entities to create a ledger of data that can be continually updated by one or more users. An initial version of the ledger is created, and then subsequent updates to the ledger reference previous

updates to the ledger. In order to accomplish this, the basic structure of a blockchain consists of three main components: data, which is the ledger update; a pointer that tells where the previous block is located; and a digest (hash) that allows one to verify that the entire contents of the previous block have not changed. Suppose that a large record of blocks has been stored in a blockchain, such as in Figure 12.2, with the final hash value not paired with any data. What happens, then, if an adversary comes along and wants to modify the data in block $k-1$? If the data in block $k-1$ is altered, then the hash contained in block k will not match the hash of the modified data in block $k-1$. This forces the adversary to have to

modify the hash in block k. But the hash of block k+1 was calculated based on the entire block k (i.e. including the k-th hash), and therefore there will now be a mismatch between the hash of block k and the hash pointer in block k+1. This process forces the adversary to continue modifying blocks until the end of the blockchain is reached. This requires a significant effort on the part of the adversary, but is possible since the adversary can calculate the hash values that he needs to replace with.
Ultimately, in order to prevent the adversary from succeeding, we require that the final hash value is stored in a manner that prevents the adversary from modifying it, thereby providing a final form of evidence preventing

modification to the blockchain.

In practice, the data contained in each block can be quite large, consisting of many data records, and therefore one will often find another hash-based data structure used in blockchains. This data structure uses hash pointers arranged in a binary tree, and is known as a **Merkle tree**. Merkle trees are useful as they make it easy for someone to prove that a certain piece of data exists within a particular block of data.

In a Merkle tree, one has a collection of n records of data that have been arranged as the leaves of a binary tree, such as shown in Figure 12.3. These data records are then grouped in pairs of two, and for each pair two hash pointers

are created, one pointing to the left data record and another to the right data record. The collection of hash pointers then serve as the data record in the next level of the tree, which are subsequently grouped in pairs of two. Again, for each pair two hash pointers are created, one pointing to the left record and the other to the right data record. We proceed up the tree until we reach the end, which is a single record that corresponds to the tree's root. The hash of this record is stored, giving one the ability to make certain that the entire collection of data records contained within the Merkle tree has not been altered.

Figure 12.3 A Merkle tree consists of data records

that have been arranged in pairs. Pairs of hash pointers point to the data records below them, and serve as a data record for the next level up in the binary tree. A final hash pointer references the head of the tree

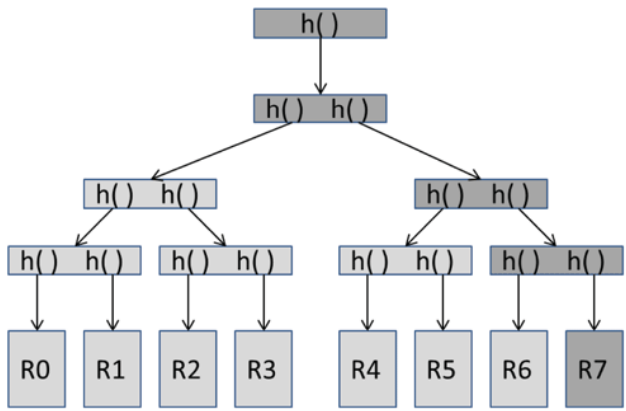


Figure 12.3 Full Alternative Text

Now suppose that someone wants to prove to you that a specific data record exists within a block. To do this, all that they need to show you is the data

record, along with the hashes on the path from the data record to the root of the Merkle tree. In particular, one does not need to show all of the other data records, one only needs to show the hashes at the higher levels. This process is efficient, requiring roughly log(n) items from the Merkle tree to be shown, where n is the number of blocks of data recorded. For example, to verify that R7 is in Figure 12.3, someone needs to show you R7, h(R7), h(R6) (but not R6), hh(R6), h(R7), the two inputs and the hash at the next level up, and the two inputs and the hash at the top. You can check these hash computations, and, since the hash function is collision-resistant, you can be sure that R7 is there and has not been changed.

Otherwise, at some level, a hash value has been changed and a collision has been found.

12.8 Exercises

1. In a family of four, what is the probability that no two people have birthdays in the same month? (Assume that all months have equal probabilities.)

2. Each person in the world flips 100 coins and obtains a sequence of length 100 consisting of Heads and Tails. (There are 2100 ≈ 1030 possible sequences.) Assume that there are approximately 1010 people in the world. What is the probability that two people obtain the same sequence of Heads and Tails? Your answer should be accurate to at least two decimal places.

3.
    1. Let EK be an encryption function with N possible keys K, N possible plaintexts, and N possible ciphertexts. Assume that if you know the encryption key K, then it is easy to find the decryption function DK (therefore, this problem does not apply to public key methods). Suppose that, for each pair (K1, K2) of keys, it is possible to find a key K3 such that EK1(EK2(m))=EK3(m) for all plaintexts

m. Assume also that for every plaintext–ciphertext pair (m, c), there is usually only one key K such that EK(m)=c. Suppose that you know a plaintext–ciphertext pair (m, c). Give a birthday attack that usually finds the key K in approximately 2N steps. (Remark: This is much faster than brute force searching through all keys K, which takes time proportional to N.)

2. Show that the shift cipher (see Section 2.1) satisfies the conditions of part (a), and explain how to attack the shift cipher mod 26 using two lists of length 6. (Of course, you could also find the key by simply subtracting the plaintext from the ciphertext; therefore, the point of this part of the problem is to illustrate part (a).)

4. Alice uses double encryption with a block cipher to send messages to Bob, so c=EK1(EK2(m)) gives the encryption. Eve obtains a plaintext–ciphertext pair (m, c) and wants to find K1, K2 by the Birthday Attack. Suppose that the output of EK has N bits. Eve computes two lists:

    1. EK(m) for $3 \cdot 2N/2$ randomly chosen keys

K.

2. DL(c) for 3·2N/2 randomly chosen keys L.

    1. Why is there a very good chance that Eve finds a key pair (L, K) such that c=EL(EK(m)) ?

    2. Why is it unlikely that (L, K) is the correct key pair? (Hint: Look at the analysis of the Meet-in-the-Middle Attack in Section 6.5.)

    3. What is the difference between the Meet-in-the-Middle Attack and what Eve does in this problem?

5. Each person who has ever lived on earth receives a deck of 52 cards and thoroughly shuffles it. What is the probability that two people have the cards in the same order? It is estimated that around 1.08×1011 people have ever lived on earth. The number of shuffles of 52 cards is 52! ≈ 8×1067.

6. Let p be a 300-digit prime. Alice chooses a secret integer k and encrypts messages by the function

$E_k(m) = mk \pmod{p}$.

1. Suppose Eve knows a cipher text c and knows the prime p. She captures Alice's encryption machine and decides to try to find m by a birthday attack. She makes two lists. The first list contains $c \cdot E_k(x)^{-1} \pmod{p}$ for some random choices of x. Describe how to generate the second list, state approximately how long the two lists should be, and describe how Eve finds m if her attack is successful.

2. Is this attack practical? Why or why not?

7. There are approximately 3×10^{147} primes with 150 digits. There are approximately 10^{85} particles in the universe. If each particle chooses a random 150-digit prime, do you think two particles will choose the same prime? Explain why or why not.

8. If there are five people in a room, what is the probability that no two of them have birthdays in the same month? (Assume that each person has probability 1/12 of being born in any given month.)

9. You use a random number generator to generate 109 random 15-digit numbers. What is the

probability that two of the numbers are equal? Your answer should be accurate enough to say whether it is likely or unlikely that two of the numbers are equal.

10. Nelson has a hash function H1 that gives an output of 60 bits. Friends tell him that this is not a big enough output, so he takes a strong hash function H2 with a 200-bit output and uses H(x)=H2(H1(x)) as his hash function. That is, he first hashes with his old hash function, then hashes the result with the strong hash function to get a 200-bit output, which he thinks is much better. The new hash function H can be computed quickly. Does it have preimage resistance, and does it have strong collision resistance? Explain your answers. (Note: Assume that computers can do up to 250 ≈ 1015 computations for this problem. Also, since it is essentially impossible to prove rigorously that most hash functions have preimage resistance or collision resistance, if your answer to either of these is “yes” then your explanation is really an explanation of why it is probably true.)

11. Bob signs contracts by signing the hash values of the contracts. He is using a hash function H with a 50-bit output. Eve has a document M that states that Bob will pay her a lot of money. Eve finds a file with 109 documents that Bob has signed. Explain how Eve can forge Bob’s signature on a

document (closely related to M) that states that Bob will pay Eve a lot of money. (Note: You may assume that Eve can do up to 230 calculations.)

12. This problem derives the formula (12.1) for the probability of at least one match in a list of length r when there are N possible birthdays.

    1. Let $f(x)=\ln(1-x)+x$ and $g(x)=\ln(1-x)+x+x2$. Show that $f'(x) \le 0$ and $g'(x) \ge 0$ for $0 \le x \le 1/2$.

    2. Using the facts that $f(0)=g(0)=0$ and f is decreasing and g is increasing, show that

       $$-x-x2 \le \ln(1-x) \le -x \quad \text{for } 0 \le x \le 1/2.$$

    3. Show that if $r \le N/2$, then

       $$-(r-1)r2N-r33N2 \le \sum j=1r-1\ln 1-jN \le -(r-1)r2N.$$

       (Hint: $\sum j=1r-1j=(r-1)r/2$ and $\sum j=1r-1j2=(r-1)r(2r-1)/6<r3/3$.)

    4. Let $\lambda=r2/(2N)$ and assume that $\lambda \le N/8$ (this implies that $r \le N/2$). Show that

       $$e-\lambda ec1/N \le \prod j=1r-11-jN \le e-\lambda ec2/N,$$

       with $c1=\lambda/2-13(2\lambda)3/2$ and $c2=\lambda/2$.

5. Observe that when N is large, ec/N is close to 1. Use this to show that as N becomes large and λ is constant with λ ≤ N/8, then we have the approximation

   $$\prod j=1r-11-jN \approx e-\lambda.$$

13. Suppose f(x) is a function with n-bit outputs and with inputs much larger than n bits (this implies that collisions must exist). We know that, with a birthday attack, we have probability 1/2 of finding a collision in approximately 2n/2 steps.

    1. Suppose we repeat the birthday attack until we find a collision. Show that the expected number of repetitions is

       $$1\cdot12+2\cdot14+3\cdot18+4\cdot116+\cdots=2$$

       (one way to evaluate the sum, call it S, is to write $S-12S=12+(2-1)14+(3-2)18+\cdots=1$).

    2. Assume that each evaluation of f takes time a constant times n. (This is realistic since the inputs needed to find collisions can be taken to have 2n bits, for example.) Show that the expected time to find a collision for the function f is a constant times n2n/2.

3. Show that the expected time to produce the messages $m0, m0', \ldots, mt-1, mt-1'$ in Section 12.2 is a constant times $tn\,2^{n/2}$.

14. Suppose we have an iterative hash function, as in Section 11.3, but suppose we adjust the function slightly at each iteration. For concreteness, assume that the algorithm proceeds as follows. There is a compression function f that operates on inputs of a fixed length. There is also a function g that yields outputs of a fixed length, and there is a fixed initial value IV. The message is padded to obtain the desired format, then the following steps are performed:

    1. Split the message M into blocks $M1, M2, \ldots, M\ell$.

    2. Let H0 be the initial value IV.

    3. For $i=1, 2, \ldots, \ell$, let $Hi=f(Hi-1, Mi||g(i))$.

    4. Let $H(M)=H\ell$.

    Show that the method of Section 12.2 can be used to produce multicollisions for this hash function.

15. Some of the steps of SRP are similar to the Diffie-

Hellman key exchange. Why not use Diffie-Hellman to log in, using the following protocol? Alice and the server use Diffie-Hellman to establish a key K. Or they could use a public key method to transmit a secret key K from the server to Alice. Then they use K, along with a symmetric system such as AES, to submit Alice's password P. Finally, the hash of the password is compared to what is stored in the computer's password file.

1. Show how Eve can do an intruder-in-the-middle attack and obtain Alice's password.

2. In order to avoid the attack in part (a), Alice and the server decide that Alice should send the hash of her password. Show that if Eve uses an intruder-in-the-middle attack, then she can log in to the server, pretending to be Alice.

3. Alice and the server have another idea. The server sends Alice a random r and Alice sends H(r||P) to the server. Show how Eve can use an intruder-in-the-middle-attack to log in as Alice.

16. Suppose that in SRP, the number u is chosen randomly by the server and sent to Alice at the

same time that s is sent. Suppose Eve has obtained v from the server's password file. Eve chooses a random a, computes $A \equiv g^{av-u} \bmod p$, and sends this value of A to the server. Then Eve computes S as $(B-3v)^a \bmod p$. Show that these computations appear to be valid to the server, so Eve can log in as Alice.

12.9 Computer Problems

1.

    1. If there are 30 people in a classroom, what is the probability that at least two have the same birthday? Compare this to the approximation given by formula (8.1).

    2. How many people should there be in a classroom in order to have a 99% chance that at least two have the same birthday? (Hint: Use the approximation to obtain an approximate answer. Then use the product, for various numbers of people, until you find the exact answer.)

    3. How many people should there be in a classroom in order to have 100% probability that at least two have the same birthday?

2. A professor posts the grades for a class using the last four digits of the Social Security number of each student. In a class of 200 students, what is the probability that at least two students have the same

four digits?

Chapter 13 Digital Signatures

For years, people have been using various types of signatures to associate their identities to documents. In the Middle Ages, a nobleman sealed a document with a wax imprint of his insignia. The assumption was that the noble was the only person able to reproduce the insignia. In modern transactions, credit card slips are signed. The salesperson is supposed to verify the signature by comparing with the signature on the card. With the development of electronic commerce and electronic documents, these methods no longer suffice.

For example, suppose you want to sign an electronic document. Why can't you simply digitize your signature and append it to the document? Anyone who has access to it can simply remove the signature and add it to something else, for example, a check for a large amount of money. With classical signatures, this would require cutting the signature off the document, or photocopying it, and pasting it on the check. This would rarely pass for an acceptable signature. However, such an electronic forgery is quite easy and cannot be distinguished from the original.

Therefore, we require that digital signatures cannot be separated from the message and attached to another. That

is, the signature is not only tied to the signer but also to the message that is being signed. Also, the digital signature needs to be easily verified by other parties. Digital signature schemes therefore consist of two distinct steps: the signing process, and the verification process.

In the following, we first present two signature schemes. We also discuss the important “birthday attacks” on signature schemes.

Note that we are not trying to encrypt the message m. In fact, often the message is a legal document, and therefore should be kept public. However, if necessary, a signed message may be encrypted after it is

signed. (This is done in PGP, for example. See Section 15.6.)

13.1 RSA Signatures

Bob has a document m that Alice agrees to sign. They do the following:

1. Alice generates two large primes p, q, and computes n=pq. She chooses eA such that $1<eA<\phi(n)$ with $\gcd(eA, \phi(n))=1$, and calculates dA such that $eAdA \equiv 1 \pmod{\phi(n)}$. Alice publishes (eA, n) and keeps private dA, p, q.

2. Alice's signature is

   $s \equiv mdA \pmod{n}$.

3. The pair (m, s) is then made public.

Bob can then verify that Alice really signed the message by doing the following:

1. Download Alice's (eA, n).

2. Calculate $z \equiv s^{eA} \pmod{n}$. If z=m, then Bob accepts the signature as valid; otherwise the signature is not valid.

Suppose Eve wants to attach Alice's signature to another message m1. She cannot simply use the pair (m1, s), since $s^{eA} \equiv m1 \pmod{n}$. Therefore, she needs s1 with $s1^{eA} \equiv m1 \pmod{n}$. This is the same problem as decrypting an RSA "ciphertext" m1 to obtain the "plaintext" s1. This is believed to be hard to do.

Another possibility is that Eve chooses s1 first, then lets the message be $m1 \equiv s1^{eA} \pmod{n}$. It does not appear that Alice can deny having signed the message m1 under the present scheme. However, it is very unlikely that m1 will be a meaningful message. It will

probably be a random sequence of characters, and not a message committing her to give Eve millions of dollars. Therefore, Alice's claim that it has been forged will be believable.

There is a variation on this procedure that allows Alice to sign a document without knowing its contents. Suppose Bob has made an important discovery. He wants to record publicly what he has done (so he will have priority when it comes time to award Nobel prizes), but he does not want anyone else to know the details (so he can make a lot of money from his invention). Bob and Alice do the following. The message to be signed is m.

1. Alice chooses an RSA modulus n (n=pq, the

product of two large primes), an encryption exponent e, and decryption exponent d. She makes n and e public while keeping p, q, d private. In fact, she can erase p, q, d from her computer's memory at the end of the signing procedure.

2. Bob chooses a random integer k (mod n) with gcd(k, n)=1 and computes $t \equiv k^e m \pmod{n}$. He sends t to Alice.

3. Alice signs t by computing $s \equiv t^d \pmod{n}$. She returns s to Bob.

4. Bob computes s/k (mod n). This is the signed message m.

Let's show that s/k is the signed message: Note that
$k^{ed} \equiv (k^e)^d \equiv k \pmod{n}$, since this is simply the encryption, then decryption, of k in the RSA scheme. Therefore,

$s/k \equiv t^d/k \equiv k^{ed}m^d/k \equiv m \pmod{n}$,

which is the signed message.

The choice of k is random, so $k^e \pmod n$ is the RSA encryption of a random number, and hence random. Therefore, $k^e m \pmod n$ gives essentially no information about m (however, it would not hide a message such as m=0). In this way, Alice knows nothing about the message she is signing.

Once the signing procedure is finished, Bob has the same signed message as he would have obtained via the standard signing procedure.

There are several potential dangers with this protocol. For example, Bob could have Alice sign a promise to pay him a million dollars. Safeguards are needed to prevent such problems. We will not

discuss these here.

Schemes such as these, called **blind signatures**, have been developed by David Chaum, who has several patents on them.

13.2 The ElGamal Signature Scheme

The ElGamal encryption method from Section 10.5 can be modified to give a signature scheme. One feature that is different from RSA is that, with the ElGamal method, there are many different signatures that are valid for a given message.

Suppose Alice wants to sign a message. To start, she chooses a large prime p and a primitive root α. Alice next chooses a secret integer a such that $1 \leq a \leq p-2$ and calculates $\beta \equiv \alpha^a \pmod{p}$. The values of p, α, and β are made public. The security of the system will be in the

fact that a is kept private. It is difficult for an adversary to determine a from (p, α, β) since the discrete log problem is considered difficult.

In order for Alice to sign a message m, she does the following:

1. Selects a secret random k with $1 \leq k \leq p-2$ such that $\gcd(k, p-1)=1$.
2. Computes $r \equiv \alpha^k \pmod{p}$ (with $0<r<p$).
3. Computes $s \equiv k^{-1}(m-ar) \pmod{p-1}$.

The signed message is the triple (m, r, s).

Bob can verify the signature as follows:

1. Download Alice's public key (p, α, β).
2. Compute $v1 \equiv \beta^{rr^s} \pmod{p}$, and $v2 \equiv \alpha^m \pmod{p}$.

3. The signature is declared valid if and only if $v1 \equiv v2 \pmod{p}$.

We now show that the verification procedure works. Assume the signature is valid. Since $s \equiv k^{-1}(m - ar) \pmod{p-1}$, we have $sk \equiv m - ar \pmod{p-1}$, so $m \equiv sk + ar \pmod{p-1}$. Therefore (recall that a congruence mod $p-1$ in the exponent yields an overall congruence mod p),

$$v2 \equiv \alpha^m \equiv \alpha^{sk+ar} \equiv (\alpha^a)^r(\alpha^k)^s \equiv \beta^r r^s \equiv v1 \pmod{p}.$$

Suppose Eve discovers the value of a. Then she can perform the signing procedure and produce Alice's signature on any desired document. Therefore, it is very important that a remain secret.

If Eve has another message m, she

cannot compute the corresponding s since she doesn't know a. Suppose she tries to bypass this step by choosing an s that satisfies the verification equation. This means she needs s to satisfy

$\beta^{r}r^{s} \equiv \alpha^{m} \pmod{p}$.

This can be rearranged to $rs \equiv \beta^{-r}\alpha^{m} \pmod{p}$, which is a discrete logarithm problem. Therefore, it should be hard to find an appropriate s. If s is chosen first, the equation for r is similar to a discrete log problem, but more complicated. It is generally assumed that it is also difficult to solve. It is not known whether there is a way to choose r and s simultaneously, though this seems to be unlikely. Therefore, the signature scheme appears to be secure,

as long as discrete logs mod p are difficult to compute (for example, p−1 should not be a product of small primes; see Section 10.2).

Suppose Alice wants to sign a second document. She must choose a new random value of k. Suppose instead that she uses the same k for messages m1 and m2. Then the same value of r is used in both signatures, so Eve will see that k has been used twice. The s values are different; call them s1 and s2. Eve knows that

$$s1k-m1 \equiv -ar \equiv s2k-m2 \pmod{p-1}.$$

Therefore,

$$(s1-s2)k \equiv m1-m2 \pmod{p-1}.$$

Let d=gcd(s1−s2, p−1). There are d

solutions to the congruence, and they can be found by the procedure given in Subsection 3.3.1. Usually d is small, so there are not very many possible values of k. Eve computes α^k for each possible k until she gets the value r. She now knows k. Eve now solves

$$ar \equiv m_1 - ks_1 \pmod{p-1}$$

for a. There are gcd(r, p−1) possibilities. Eve computes α^a for each one until she obtains β, at which point she has found a. She now has completely broken the system and can reproduce Alice's signatures at will.

Example

Alice wants to sign the message $m_1 = 151405$ (which corresponds to *one*,

if we let 01=a, 02=b, …). She chooses p=225119. Then $\alpha=11$ is a primitive root. She has a secret number a. She computes $\beta \equiv \alpha^a \equiv 18191 \pmod{p}$. To sign the message, she chooses a random number k and keeps it secret. She computes $r \equiv \alpha^k \equiv 164130 \pmod{p}$. Then she computes

$$s1 \equiv k^{-1}(m1 - ar) \equiv 130777 \pmod{p-1}.$$

The signed message is the triple (151405, 164130, 130777).

Now suppose Alice also signs the message m2=202315 (which is *two*) and produces the signed message (202315, 164130, 164899). Immediately, Eve recognizes that Alice used the same value of k, since the value of r is the same in both signatures. She

therefore writes the congruence

$$-34122k \equiv (s1-s2)k \equiv m1-m2 \equiv -50910 \pmod{p-1}.$$

Since gcd(−34122, p−1)=2, there are two solutions, which can be found by the method described in Subsection 3.3.1. Divide the congruence by 2:

$$-17061k \equiv -25455 \pmod{(p-1)/2}.$$

This has the solution $k \equiv 239 \pmod{(p-1)/2}$, so there are two values of $k \pmod p$, namely 239 and 239+(p −1)/2=112798. Calculate

$$\alpha 239 \equiv 164130, \quad \alpha 112798 \equiv 59924 \pmod p.$$

Since the first is the correct value of r, Eve concludes that k=239. She now rewrites $s1k \equiv m1-ar \pmod{p-1}$ to obtain

$$164130a \equiv ra \equiv m1 - s1k \equiv 187104 \pmod{p-1}.$$

Since gcd(164130, p−1)=2, there are two solutions, namely a=28862 and a=141421, which can be found by the method of Subsection 3.3.1. Eve computes

$$\alpha^{28862} \equiv 206928, \quad \alpha^{141421} \equiv 18191 \pmod{p}.$$

Since the second value is β, she has found that a=141421.

Now that Eve knows a, she can forge Alice's signature on any document.

The ElGamal signature scheme is an example of a **signature with appendix**. The message is not easily recovered from the signature (r, s). The message m must be included in the verification procedure. This is in contrast to the RSA

signature scheme, which is a **message recovery scheme**. In this case, the message is readily obtained from the signature s. Therefore, only s needs to be sent since anyone can deduce m as seA (mod n). It is unlikely that a random s will yield a meaningful message m, so there is little danger that someone can successfully replace a valid message with a forged message by changing s.

13.3 Hashing and Signing

In the two signature schemes just discussed, the signature can be longer than the message. This is a disadvantage when the message is long. To remedy the situation, a hash function is used. The signature scheme is then applied to the hash of the message, rather than to the message itself.

The hash function h is made public. Starting with a message m, Alice calculates the hash h(m). This output h(m) is significantly smaller, and hence signing the hash may be done more quickly than signing the entire message. Alice calculates the signed message

sig(h(m)) for the hash function and uses it as the signature of the message. The pair (m, sig(h(m))) now conveys basically the same knowledge as the original signature scheme did. It has the advantages that it is faster to create (under the reasonable assumption that the hash operation is quick) and requires less resources for transmission or storage.

Is this method secure? Suppose Eve has possession of Alice's signed message (m, sig(h(m)). She has another message m' to which she wants to add Alice's signature. This means that she needs $sig(h(m')) = sig(h(m))$; in particular, she needs $h(m')=h(m)$. If the hash function is one-way, Eve will find it hard to find

any such m′. The chance that her desired m′ will work is very small. Moreover, since we require our hash function to be strongly collision-resistant, it is unlikely that Eve can find two messages m1 ≠ m2 with the same signatures. Of course, if she did, she could have Alice sign m1, then transfer her signature to m2. But Alice would get suspicious since m1 (and m2) would very likely be meaningless messages.

In the next section, however, we'll see how Eve can trick Alice if the size of the message digest is too small (and we'll see that the hash function will not be strongly collision-resistant, either).

13.4 Birthday Attacks on Signatures

Alice is going to sign a document electronically by using one of the signature schemes to sign the hash of the document. Suppose the hash function produces an output of 50 bits. She is worried that Fred will try to trick her into signing an additional contract, perhaps for swamp land in Florida, but she feels safe because the chance of a fraudulent contract having the same hash as the correct document is 1 out of 250, which is approximately 1 out of 1015. Fred can try several fraudulent contracts, but it is very unlikely that he can find one that has the right hash.

Fred, however, has studied the birthday attack and does the following. He finds 30 places where he can make a slight change in the document: adding a space at the end of a line, changing a wording slightly, etc. At each place, he has two choices: Make the small change or leave the original. Therefore, he can produce 230 documents that are essentially identical with the original. Surely, Alice will not object to any of these versions. Now, Fred computes the hash of each of the 230 versions and stores them. Similarly, he makes 230 versions of the fraudulent contract and stores their hashes. Consider the generalized birthday problem with r=230 and N=250. The probability of a match is approximately

$1-e-r2/N=1-e-1024 \approx 1.$

Therefore, it is very likely that a version of the good document has the same hash as a version of the fraudulent contract. Fred finds the match and asks Alice to sign the good version. He plans to append her signature to the fraudulent contract, too. Since they have the same hash, the signature would be valid for the fraudulent one, so Fred could claim that Alice agreed to buy the swamp land.

But Alice is an English teacher and insists on removing a comma from one sentence. Then she signs the document, which has a completely different hash from the document Fred asked her to sign. Fred is foiled again. He now is

faced with the prospect of trying to find a fraudulent contract that has the same hash as this new version of the good document. This is essentially impossible.

What Fred did is called the birthday attack. Its practical implication is that you should probably use a hash function with output twice as long as what you believe to be necessary, since the birthday attack effectively halves the number of bits. What Alice did is a recommended way to foil the birthday attack on signature schemes. Before signing an electronic document, make a slight change.

13.5 The Digital Signature Algorithm

The National Institute of Standards and Technology proposed the Digital Signature Algorithm (DSA) in 1991 and adopted it as a standard in 1994. Later versions increased the sizes of the parameters. Just like the ElGamal method, DSA is a digital signature scheme with appendix. Also, like other schemes, it is usually a message digest that is signed. In this case, let's say the hash function produces a 256-bit output. We will assume in the following that our data message m has already been hashed. Therefore, we are trying to sign a 256-bit message.

The generation of keys for DSA proceeds as follows. First, there is an initialization phase:

1. Alice finds a prime q that is 256 bits long and chooses a prime p that satisfies $q|p-1$ (see Exercise 15). The discrete log problem should be hard for this choice of p. (In the initial version, p had 512 bits. Later versions of the standard require longer primes, for example, 2048 bits.)

2. Let g be a primitive root mod p and let $\alpha \equiv g^{(p-1)/q} \pmod{p}$. Then $\alpha^q \equiv 1 \pmod{p}$.

3. Alice chooses a secret a such that $1 \leq a < q-1$ and calculates $\beta \equiv \alpha^a \pmod{p}$.

4. Alice publishes (p, q, α, β) and keeps a secret.

Alice signs a message m by the following procedure:

1. Select a random, secret integer k such that $0 < k < q-1$.

2. Compute $r = (\alpha^k \pmod{p}) \pmod{q}$.

3. Compute $s \equiv k^{-1}(m+ar) \pmod{q}$.

4. Alice's signature for m is (r, s), which she sends to Bob along with m.

For Bob to verify, he must

1. Download Alice's public information (p, q, α, β).

2. Compute $u1 \equiv s^{-1}m \pmod{q}$, and $u2 \equiv s^{-1}r \pmod{q}$.

3. Compute $v=(\alpha^{u1}\beta^{u2} \pmod{p}) \pmod{q}$.

4. Accept the signature if and only if v=r.

We show that the verification works. By the definition of s, we have

$$m \equiv (-ar+ks) \pmod{q},$$

which implies

$$s^{-1}m \equiv (-ars^{-1}+k) \pmod{q}.$$

Therefore,

$k \equiv s^{-1}m + ars^{-1} \pmod{q} \equiv u1 + au2 \pmod{q}$.

So

$$\alpha^k = \alpha^{u1+au2} = (\alpha^{u1}\beta^{u2} \pmod{p}) \pmod{q}.$$

Thus $v = r$.

As in the ElGamal scheme, the integer a must be kept secret. Anyone who has knowledge of a can sign any desired document. Also, if the same value of k is used twice, it is possible to find a by the same procedure as before.

In contrast to the ElGamal scheme, the integer r does not carry full information about k. Knowing r allows us to find only the mod q value. There are approximately $2^{2048-256} = 2^{1792}$ numbers mod p that reduce to a given number mod q.

What is the advantage of having $\alpha^q \equiv 1 \pmod{p}$ rather than using a primitive root? Recall the Pohlig-Hellman attack for solving the discrete log problem. It could find information mod small prime factors of $p-1$, but it was useless mod large prime factors such as q. In the ElGamal scheme, an attacker could determine $a \pmod{2t}$, where $2t$ is the largest power of 2 dividing $p-1$. This would not come close to finding a, but the general philosophy is that many little pieces of information collectively can often be useful. The DSA avoids this problem by removing all but the mod q information for a.

In the ElGamal scheme, three modular

exponentiations are needed in the verification step. This step is modified for the DSA so that only two modular exponentiations are needed. Since modular exponentiation is one of the slower parts of the computation, this change speeds up the verification, which can be important if many signatures need to be verified in a short time.

13.6 Exercises

1. Show that if someone discovers the value of k used in the ElGamal signature scheme, then a can also be determined if gcd(r, p−1) is small.

2. Alice signs the hash of a message. Suppose her hash function satisfies $h(x) \equiv 2x \pmod{101}$ and $1 \leq h(x) \leq 100$ for all x. Suppose m is a valid signed message from Alice. Give another message m1 for which the same signature is also valid.

3. Alice says that she is willing to sign a petition to save koala bears. Alice's signing algorithm uses a hash function that has an output of 60 bits (and she signs the hash of the document). Describe how Eve can trick Alice into signing a statement allowing Eve unlimited withdrawals from Alice's bank account.

4. Alice uses RSA signatures (without a hash function).

    1. Eve wants to produce Alice's signature on the document m=123456789. Why is this difficult? Explain this by saying what

difficult cryptographic problem must be solved. (Do not say that it's because Eve does not know the decryption exponent d. Why isn't there another way to produce the signature?)

2. Since part (a) is too hard for Eve, she decides to produce Alice's valid RSA signature on a document so that the signature is s=112090305 (= Alice, when A=01, L=12, etc.). How does Eve find an appropriate message?

5. Nelson thinks he has a new version of the signature scheme. He chooses RSA parameters n, e, and d. He signs by computing $s \equiv m^d \pmod{n}$. The verification equation is $(s-m)(s+m) \stackrel{?}{=} s^2 - m^2$.

    1. Show that if Nelson correctly follows the signing procedure, or if he doesn't, then the signature is declared valid.

    2. Show that Eve can forge Nelson's signature on any document m, even though she does not know d.

    (The point of this exercise is that the verification equation is important. All Eve needs to do is satisfy the verification equation. She does not need

to follow the prescribed procedure for producing the signature.)

6. Alice has a long message m. She breaks m into blocks of 256 bits: $m=m1\|m2\|\cdots\|mN$. She regards each block as a number between 0 and $2256-1$, and she signs the sum $t=m1+m2+\cdots+mN$. This means her signed message is (m, sig(t)), where *sig* is the signing function. Is this a good idea? Why or why not?

7. Suppose that (m, r, s) is a message signed with the ElGamal signature scheme. Choose h with $\gcd(h, p-1)=1$ and let $r1 \equiv r^h \pmod p$. Let $s1 \equiv sr1h-1r-1 \pmod{p-1}$.

    1. Find a message m1 for which (m1, r1, s1) is a valid signature.

    2. This method allows Eve to forge a signature on the message m1. Why is it unlikely that this causes problems?

8. Let $p=11$, $q=5$, $\alpha=3$, and $k=3$. Show that $(\alpha^k \pmod p) \pmod q \neq (\alpha^k \pmod q) \pmod p$. This shows that the order of operations in the DSA is important.

9. There are many variations to the ElGamal digital signature scheme that can be obtained by altering

the signing equation $s \equiv k^{-1}(m - ar) \pmod{p-1}$. Here are some variations.

1. Consider the signing equation $s \equiv a^{-1}(m - kr) \pmod{p-1}$. Show that the verification $\alpha^m \equiv (\alpha^a)^s r^r \pmod{p}$ is a valid verification procedure.

2. Consider the signing equation $s \equiv am + kr \pmod{p-1}$. Show that the verification $\alpha^s \equiv (\alpha^a)^m r^r \pmod{p}$ is a valid verification procedure.

3. Consider the signing equation $s \equiv ar + km \pmod{p-1}$. Show that the verification $\alpha^s = (\alpha^a)^r r^m \pmod{p}$ is a valid verification procedure.

10. Consider the following variant of the ElGamal Signature Scheme: Alice chooses a large prime p, a primitive root α, and a secret integer a. She computes $h \equiv \alpha^a \pmod{p}$. The numbers p, α, h are made public and a is kept secret. If $m < p-1$, Alice signs m as follows: She chooses a random integer k and computes $r \equiv \alpha^k \pmod{p}$, with $0 < r < p-1$, and $s \equiv am - kr \pmod{p-1}$. The signed message is (m, r, s). Bob verifies the signature by checking that $\alpha^s r^r \equiv h^m \pmod{p}$. If $m \geq p-1$, she breaks m into blocks and signs each block.

1. Show that if Alice signs correctly then the verification congruence is satisfied.

2. Suppose Eve has a document m1 and she wants to forge Alice's signature on m1. That is, she wants to find r1 and s1 such that (m1, r1, s1) is valid. Eve chooses r1=2015 and tries to find a suitable s1. Why will it probably be hard to find s1?

3. Suppose Alice has a very long message m and wants to decrease the size of the signature. How can she use a hash function to do this? Explicitly give the modifications of the above equations that must be done to accomplish this.

11. The ElGamal signature scheme presented is weak to a type of attack known as existential forgery. Here is the basic existential forgery attack. Choose u, v such that gcd(v, p−1)=1. Compute $r \equiv \beta^v \alpha^u \pmod{p}$ and $s \equiv -rv^{-1} \pmod{p-1}$.

    1. Prove the claim that the pair (r, s) is a valid signature for the message $m \equiv su \pmod{p-1}$ (of course, it is likely that m is not a meaningful message).

    2. Suppose a hash function h is used and the signature must be valid for h(m) instead

of for m (so we need to have h(m)=su). Explain how this scheme protects against existential forgery. That is, explain why it is hard to produce a forged, signed message by this procedure.

12. Alice's RSA public key is (n, e) and her private key is d. Recall that a document with an RSA signature (m, s) is valid if $m \equiv s^e \pmod{n}$. Bob wants Alice to sign a document m but he does not want Alice to read the document. Assume $m<n$. They do the following:

    1. Bob chooses a random integer k with gcd(k, n)=1. He computes $m1 \equiv km \pmod{n}$.

    2. Alice signs m1 by computing $s1 \equiv m1^d \pmod{n}$.

    3. Bob divides s1 by k mod n to obtain $s \equiv k^{-1}s1 \pmod{n}$.

    1. Show that (m, s) is valid.

    2. Why is it assumed that gcd(k, n)=1?

13. Alice wants to sign a document using the ElGamal signature scheme. Suppose her random number

generator is broken, so she uses k=a in the signature scheme. How will Eve notice this and how can Eve determine the values of k and a (and thus break the system)?

14. Suppose Alice signs contracts using a 30-bit hash function h (and h is known to everyone). If m is the contract, then (m, sig(h(m))) is the signed contract (where *sig* is some public signature function). Eve has a file of 220 fraudulent contracts. She finds a file with 220 contracts with valid signatures (by Alice) on them. Describe how Eve can accomplish her goal of putting Alice's signature on at least one fraudulent document.

15.

    1. In several cryptographic protocols, one needs to choose a prime p such that q=(p −1)/2 is also prime. One way to do this is to choose a prime q at random and then test 2q+1 for primality. Suppose q is chosen to have approximately 300 decimal digits. Assume 2q+1 is a random odd integer of 300 digits. (This is not quite accurate, since 2q+1 cannot be congruent to 1 mod 3, for example. But the assumption is good enough for a rough estimate.) Show that the probability that 2q+1 is prime is approximately 1/345 (use the prime

number theorem, as in Section 9.3). This means that with approximately 345 random choices for the prime q, you should be able to find a suitable prime p.

2. In a version of the Digital Signature Algorithm, Alice needs a 256-bit prime q and a 2048-bit prime p such that $q|p-1$. Suppose Alice chooses a random 256-bit prime q and a random 1792-bit even number k such that $qk+1$ has 2048 bits. Show that the probability that $qk+1$ is prime is approximately 1/710. This means that Alice can find a suitable q and p fairly quickly.

16. Consider the following variation of the ElGamal signature scheme. Alice chooses a large prime p and a primitive root α. She also chooses a function $f(x)$ that, given an integer x with $0 \le x < p$, returns an integer $f(x)$ with $0 \le f(x) < p-1$. (For example, $f(x) = x^7 - 3x + 2 \pmod{p-1}$ for $0 \le x < p$ is one such function.) She chooses a secret integer a and computes $\beta \equiv \alpha^a \pmod{p}$. The numbers p, α, β and the function $f(x)$ are made public.

    Alice wants to sign a message m:

    1. Alice chooses a random integer k with

gcd(k, p−1)=1.

2. She computes $r \equiv \alpha^k \pmod{p}$.

3. She computes $s \equiv k^{-1}(m - f(r)a) \pmod{p-1}$.

The signed message is (m, r, s).

Bob verifies the signature as follows:

1. He computes $v1 \equiv \beta^{f(r)} r^s \pmod{p}$.

2. He computes $v2 \equiv \alpha^m \pmod{p}$.

3. If $v1 \equiv v2 \pmod{p}$, he declares the signature to be valid.

1. Show that if all procedures are followed correctly, then the verification equation is true.

2. Suppose Alice is lazy and chooses the constant function satisfying f(x)=0 for all x. Show that Eve can forge a valid signature on every message m1 (for example, give a value of k and of r and s that will give a valid signature for the message m1).

17. Alice wants to sign a long message $m=m1\|m2\|\cdots\|mL$ that she has broken into blocks m1, …, mL. She knows that signing each block mi individually is wasteful, so she computes $g(m)=m1\oplus m2\oplus\cdots\oplus mL$ and signs g(m). Her signed message is (m, sigA(g(m))). Suppose Eve has a message that Alice signed. How can Eve put Alice's signature on fraudulent messages?

18. In some scenarios, it is necessary to have a digital document signed by multiple participants. For example, a contract issued by a company might need to be signed by both the Issuer and the Supervisor before it is valid. To accomplish this, a trusted entity chooses n=pq to be the product of two large, distinct primes and chooses integers k1, k2, k3>1 with $k1k2k3 \equiv 1 \pmod{(p-1)(q-1)}$. The pair (n, k1) is given to the Issuer, the pair (n, k2) is given to the Supervisor, and the pair (n, k3) is made public.

    1. Under the assumption that RSA is hard to decrypt, why should it be difficult for someone who knows at most one of k1, k2 to produce s such that $sk3 \equiv m \pmod{n}$?

    2. Devise a procedure where the Issuer signs the contract first and gives the signed contract to the Supervisor, who

then signs it in such a way that anyone can verify that the document was signed by both the Issuer and the Supervisor. Use part (a) to show why the verification convinces someone that both parties signed the contract.

13.7 Computer Problems

1. Suppose we use the ElGamal signature scheme with p=65539, α=2, β=33384. We send two signed messages (m, r, s):

   (809, 18357, 1042) (=
   hi) and (22505, 18357, 26272) (= bye).

   1. Show that the same value of k was used for each signature.

   2. Use this fact to find this value of k and to find the value of a such that $\beta \equiv \alpha a \pmod{p}$.

2. Alice and Bob have the following RSA parameters:

   nA=171024704183616109700818066925197841516671277, nB=83

   Bob knows that

   pB=9876345769783456893461 3, qB=8495789457893457345793

   (where nB=pBqB). Alice signs a document and sends the document and signature (m, s) (where

s ≡ mdA (mod nA)) to Bob. To keep the contents of the document secret, she encrypts using Bob's public key. Bob receives the encrypted signature pair (m1, s1) ≡ (meB, seB) (mod nB), where

m1=4187265539970942585779800550613051509405479 56s1=749

Find the message m and verify that it came from Alice. (The numbers m1, s1, nA, nB, pB, qB are stored as *sigpairm1, sigpairs1, signa, signb, sigpb, sigqb* in the downloadable computer files (`bit.ly/2JbcS6p`).)

3. In problem 2, suppose that Bob had primes pB=7865712896579 and qB=8495789457893457345793. Assuming the same encryption exponents, explain why Bob is unable to verify Alice's signature when she sends him the pair (m2, s2) with

m2=1482376523249871234451241871713 0930, s2=431761216284

What modifications need to be made for the procedure to work? (The numbers m2 and s2 are stored as *sigpairm2, sigpairs2* in the downloadable computer files (`bit.ly/2JbcS6p`).)

Chapter 14 What Can Go Wrong

The mathematics behind cryptosystems is only part of the picture. Implementation is also very important. The best systems, when used incorrectly, can lead to serious problems. And certain design considerations that might look good at the time can later turn out to be bad.

We start with a whimsical example that (we hope) has never been used in practice. Alice wants to send a message to Bob over public channels. They decide to use the three-pass protocol (see Section 3.6).

Here is a physical description. Alice puts her message in a box, puts on her lock, and send the box to Bob. Bob puts on his lock and sends the box back to Alice, who takes her lock off the box and sends the box to Bob. He takes off his lock and opens the box to retrieve the message. Notice that the box always is locked when it is in transit between Alice and Bob.

In Section 3.6, we gave an implementation using modular exponentiation. But Alice and Bob know that the one-time pad is the most secure cryptosystem, and they want to use only the best. Therefore, Alice and Bob each choose their own one-time pads, call them KA and KB. Alice's

message is M. She encrypts it as $C1=M \oplus KA$ and sends C1 to Bob, who computes $C2=C1 \oplus KB$ and sends C2 back to Alice. Then Alice removes her "lock" KA by computing $C3=C2 \oplus KA$ and sends C3 to Bob. He removes his lock by computing $M=C3 \oplus KB$.

Meanwhile, Eve intercepts C1, C2, C3 and computes

$$C1 \oplus C2 \oplus C3=(M \oplus KA) \oplus (M \oplus KA \oplus KB) \oplus (M \oplus KB)=M.$$

The moral of the story: even the best cryptosystem can be insecure if not used appropriately.

In this chapter, we describe some situations where mistakes were made, leading to security flaws.

14.1 An Enigma “Feature”

For its time, the Enigma machine (see Section 2.7) was an excellent system, and the German troops’ belief in this led to security breaches. After all, if your post is being overrun or your ship is sinking, are you going to try to escape, or are you going to risk your life to destroy a crypto machine that you’ve been told is so good that no one can break the system?

But there were also other lapses in its use. One of the most famous took advantage of an inherent “feature” of Enigma. The design, in particular the reflector (see Section 2.7), meant that a

letter could not be encrypted as itself. So an A would never encrypt as an A, and a B would never encrypt as a B, etc. This consequence of the internal wiring probably looked great to people unfamiliar with cryptography. After all, there would always be “complete encryption." The plaintext would always be completely changed. But in practice it meant that certain guesses for the plaintext could immediately be discarded as impossible.

One day in 1941, British cryptographers intercepted an Enigma message sent by the Italian navy, and Mavis Batey, who worked at Bletchley Park, noticed that the ciphertext did not contain the letter L. Knowing the feature of Enigma, she

guessed that the original message was simply many repetitions of L (maybe a bored radio operator was tapping his finger on the keyboard while waiting for the next message). Using this guess, it was possible to determine the day's Enigma key. One message was "Today's the day minus three." This alerted the British to a surprise attack on their Mediterranean fleet by the Italian navy. The resulting Battle of Cape Matapan established the dominance of the British fleet in the eastern Mediterranean.

14.2 Choosing Primes for RSA

The security of the RSA cryptosystem (see Chapter 9) relies heavily on the inability of an attacker to factor the modulus n=pq. Therefore, it is very important that p and q be chosen in an unpredictable way. This usually requires a good pseudorandom number generator to give a starting point in a search for each prime, and a pseudorandom number generator needs a random seed, or some random data, as an input to start its computation.

In 1995, Ian Goldberg and David Wagner, two computer science graduate

students at the University of California at Berkeley, were reading the documentation for the Netscape web browser, one of the early Internet browsers. When a secure transaction was needed, the browser generated an RSA key. They discovered that a time stamp was used to form the seed for the pseudorandom number generator that chose the RSA primes. Using this information, they were able to deduce the primes used for a transaction in just a few seconds on a desktop computer.

Needless to say, Netscape quickly put out a new version that repaired this flaw. As Jeff Bidzos, president of RSA Data Security pointed out, Netscape "declined to have us help them on the

implementation of our software in their first version. But this time around, they've asked for our help" (*NY Times*, Sept. 19, 1995).

Another potential implementation flaw was averted several years ago. A mathematician at a large electronics company (which we'll leave nameless) was talking with a colleague, who mentioned that they were about to put out an implementation of RSA where they saved time by choosing only one random starting point to look for primes, and then having p and q be the next two primes larger than the start. In horror, the mathematician pointed out that finding the next prime after n, a very straightforward process, immediately

yields the larger prime, thus breaking the system.

Perhaps the most worrisome problem was discovered in 2012. Two independent teams collected RSA moduli from the web, for example from X-509 certificates, and computed the gcd of each pair (see [Lenstra2012 et al.] and [Heninger et al.]). They found many cases where the gcd gave a nontrivial factor. For example, the team led by Arjen Lenstra collected 11.4×106 RSA moduli. They computed the gcd of each pair of moduli and found 26965 moduli where the gcd gave a nontrivial factor. Fortunately, most of these moduli were at the time no longer in use, but this is still a serious security

problem. Unless the team told you, there is no way to know whether your modulus is one of the bad ones unless you want to compute the gcd of your modulus with the 6.4×106 other moduli (this is, of course, much faster than computing the gcd of all pairs, not just those that include your modulus). And if you find that your modulus is bad, you have factored someone else's modulus and thus have broken their system.

How could this have happened? Probably some bad pseudorandom number generators were used. Let's suppose that your pseudorandom number generator can produce only 1 million different primes (this could easily be the case if you don't have a

good source of random seeds; [Heninger et al.] gives a detailed analysis of this situation). You use it to generate 2000 primes, which you pair into 1000 RSA moduli. So what could go wrong?

Recall the Birthday Paradox (see Section 12.1). The number of "birthdays" is N=106 and the number of "people" is 2000=2N. It is likely that two "people" have the same "birthday." That is, two of the primes are equal (it is very unlikely that they are used for the same modulus, especially if the generating program is competently written). Therefore, two moduli will share a common prime, and this can be discovered by computing their gcd.

Of course, there are enough large primes

that a good pseudorandom number generator will potentially produce so many primes that is is unlikely that a prime will be repeated. As often is the case in cryptography, we see that security ultimately relies on the quality of the pseudorandom number generator. Generating randomness is hard.

14.3 WEP

As wireless technology started to replace direct connections in the 1990s, the WEP (Wired Equivalent Privacy) Algorithm was introduced and was the standard method used from 1997 to 2004 for users to access the Internet via wireless devices via routers. The intent was to make wireless communication as secure as that of a wired connection. However, as we'll see, there were several design flaws, and starting in 2004 it was replaced by the more secure WPA (Wi-Fi Protected Access), WPA2, and WPA3.

The basic arrangement has a central

access point, usually a router, and several laptop computers that want to access the Internet through the router. The idea is to make the communications between the laptops and the router secure. Each laptop has the same WEP key as the other users for this router.

A laptop initiates communication with the router. Here is the protocol.

1. The laptop sends an initial greeting to the router.

2. Upon receiving the greeting, the router generates a random 24-bit IV (= initial value) and sends it to the laptop.

3. The laptop produces a key for RC4 (a stream cipher described in Section 5.3) by concatenating the WEP key for this router with the IV received from the router:

   RC4Key = WEPKey||IV.

   and uses it to produce the bitstream *RC4*.

4. The laptop also computes the checksum ch=CRC32(Message). (See the description of CRC-32 at the end of this section.)

5. The laptop forms the ciphertext

   C= RC4 ⊕ (Message, ch),

   which it sends to the router along with the IV (so the router does not need to remember what IV it sent).

6. The router uses the WEPKey and the IV to form

   RC4Key=WEPKey||IV,

   and then uses this to produce the bitstream RC4.

7. The router computes C⊕ RC4 =(message, ch).

8. If ch=CRC32(Message), the message is regarded as authentic and is sent to the Internet. If not, it is rejected.

Notice that, except for the authentication step, this is essentially a one-time pad, with RC4 supplying the pseudorandom bitstream.

Usually, the WEP key length was 40 bits. Why so short? This meant there were 240 ≈ 1012 possible keys, so a brute force attack could find the key. But back when the algorithm was developed, U.S. export regulations prohibited the export of more secure cryptography, so the purpose of 40 bits was to allow international use of WEP. Later, many applications changed to 104 bits. However, it should be emphasized that WEP is considered insecure for any key size because of the full collection of design flaws in the protocol, which is why the Wi-Fi Alliance subsequently developed new security protocols to protect wireless communications.

The designers of WEP knew that

reusing a one-time pad is insecure, which is why they included the 24-bit IV. Since the IV is concatenated with the 40-bit WEP key to form the 64-bit RC4Key (or 24+104=128 bits in more recent versions), there should be around 224 ≈ 1.7×107 keys used to generate RC4 bitstreams. This might seem secure, but a highly used router might be expected to use almost every possible key in a short time span.

But the situation is even worse! The Birthday Paradox (see Section 12.1) enters again. There are N=224 possible IV values, and N=212=4096. Therefore, after, for example, 10000 communications, it is very likely that some IV will have been used twice.

Since the IV is sent unencrypted from the router to the laptop, this is easy for Eve to notice. There are now two messages encrypted with the same pseudorandom one-time pad. This allows Eve to recover these two messages (see Section 4.3). But Eve also obtains the RC4 bitstream used for encryption. The IV and this bitstream are all that Eve needs in Step (e) of the protocol to do the encryption. The WEP key is not required once the bitstream corresponding to the IV is known. Therefore, Eve has gained access to the router and can send messages as desired.

There is even more damage that Eve can do. She intercepts a ciphertext $C = \text{RC4} \oplus (\text{Message}, \text{ch})$, and we are

not assuming this RC4 was obtained from a repeated IV value. So the C is rather securely protecting the message. But suppose Eve guesses correctly that the message says

Send to IP address 69.72.169.241. Here is my credit card number, etc.

Eve's IP address is 172.16.254.1. Let M1 be the message consisting of all 0s except for the XOR of the two IP addresses in the appropriate locations. Then

M2=M⊕M1=Send to IP address 172.16.254.1. Here is my credit card number, etc.

Moreover, because of the nature of CRC-32,

ch(M2)=ch(M)⊕ch(M1).

Eve doesn't know M, and therefore,

doesn’t know M2. But she does know ch(M1). Therefore, she takes the original C=RC4⊕(M, ch(M)) and forms

C⊕(M1, ch(M1))=RC4⊕(M, ch(M))⊕(M1, ch(M1)).

Since M⊕M1=M2 and

ch(M)⊕ch(M1)=ch(M⊕M1)=ch(M2),

she has formed

RC4⊕(M2, ch(M2)).

This will be accepted by the router as an authentic message. The router forwards it to the Internet and it is delivered to Eve’s IP address. Eve reads the message and obtains the credit card number.

This last flaw easily could have been avoided if a cryptographic hash function had been used in place of CRC-32. Such

functions are highly non-linear (so it is unlikely that $ch(M \oplus M1) = ch(M) \oplus ch(M1)$), and it would be very hard to figure out how to modify the checksum so that the message is deemed authentic.

The original version of WEP used what is known as Shared Key Authentication to control access. In this version, after the laptop initiates the communication with a greeting, the router sends a random challenge bitstring to the laptop. The laptop encrypts this using the WEP key as the key for RC4 and then XORing the challenge with the RC4 bitstring:

$RC4(WEPKey) \oplus Challenge.$

This is sent to the router, which can do

the same computation and compare results. If they agree, access is granted. If not, the laptop is rejected.

But Eve sees the Challenge and also RC4(WEPKey) $\oplus$ Challenge. A quick XOR of these two strings yields RC4(WEPKey). Now Eve can respond to any challenge, even if she does not know the WEP key, and thereby gain access to the router.

This access method was soon dropped and replaced with the open access system described above, where anyone can try to send a message to the system, but only the ones that decrypt and yield a valid checksum are accepted.

14.3.1 CRC-32

CRC-32 (= 32-bit Cyclic Redundancy Check) was developed as a way to detect bit errors in data. The message M is written in binary and these bits are regarded as the coefficients of a polynomial mod 2. For example, the message `10010101` becomes the polynomial x7+x4+x2+1. Divide this polynomial by the polynomial

p(x)=x32+x26+x23+x22+x16+x12+x11+x10+x8+x7+x5+x4+x2+x+1.

Let r(x) be the remainder, which is a polynomial of degree at most 31. Reduce the coefficients of r(x) mod 2 to obtain a binary string of length 32 (if the degree is less than 31, the binary string will start with some 0s corresponding to the 0-coefficients for the high degree

terms). For an example of such polynomial division, see Section 3.11. This binary string is the checksum ch(M).

Adding two polynomials mod 2 corresponds to computing the XOR of the corresponding bitstrings formed from their coefficients. For example, the polynomial x7+x4+x2+1 corresponds to `10010101` and x4+x+1 corresponds to `00010011`, with a few 0s added to match the first string. The sum of the two polynomials mod 2 is x7+x2+x, which corresponds to `10000110` = `10010101`⊕`00010011`. Since dividing the sum of two polynomials by p(x) yields a remainder that is the sum of the remainders that would be obtained by

dividing each polynomial individually, it can be deduced that

$$ch(M1 \oplus M2)=ch(M1) \oplus ch(M2)$$

for any messages M1, M2.

14.4 Exercises

1. The Prime Supply Company produces RSA keys. It has a database of 108 primes with 200 decimal digits and a database of 1010 primes with 250 decimal digits. Whenever it is asked for an RSA modulus, it randomly chooses a 200-digit prime p and a 250-digit prime q from its databases, and then computes n=pq. It advertises that it has 1018 possible moduli to distribute and brags about its great level of security. After the Prime Supply Company has used this method to supply RSA moduli to 20000 customers, Eve tries computing gcd's of pairs of moduli of these clients (that is, for each pair of clients, with RSA moduli n' and n'', she computes $\gcd(n', n'')$). What is the likely outcome of Eve's computations? Explain.

2. The Modulus Supply Company sells RSA moduli. To save money, it has one 300-digit prime p and, for each customer, it randomly chooses another 300-digit prime q (different from p and different from q supplied to other customers). Then it sells n=pq, along with encryption and decryption exponents, to unsuspecting customers.

1. Suppose Eve suspects that the company is using this method of providing moduli to customers. How can she read their messages? (As usual, the modulus n and the encryption exponent e for each customer are public information.)

2. Now suppose that the customers complain that Eve is reading their messages. The company computes a set S of 106 primes, each with 300 digits. For each customer, it chooses a random prime p from S, then randomly chooses a 300-digit prime q, as in part (a). The 100 customers who receive moduli n=pq from this update are happy and Eve publicly complains that she no longer can break their systems. As a result, 2000 more customers buy moduli from the company. Explain why the 100 customers probably have distinct primes p, but among the 2000 customers there are probably two with the same p.

3. Suppose p(x)=x4+x+1 is used in place of the p(x) used in CRC-32. If S is a binary string, then express it as a polynomial f(x), divide by p(x), and let r(x) be the remainder mod 2. Change this back to a binary string of length 4 and call this string

c(S). This produces a check-sum in a manner similar to CRC-32.

1. Compute c(1001001) and c(1101100).

2. Compute c(1001001 ⊕ 1101100) and show that it is the XOR of the two individual XORs obtained in part (a).

4. You intercept the ciphertext *TUCDZARQKERUIZCU*, which was encrypted using an Enigma machine. You know the plaintext was either ATTACKONTHURSDAY or ATTACKONSATURDAY. Which is it?

5.

1. You play the CI Game from Chapter 4 with Bob. You give him the plaintexts CAT and DOG. He chooses one of these at random and encrypts it on his Enigma machine, producing the ciphertext DDH. Can you decide which plaintext he encrypted?

2. Suppose you play the CI Game in part (a) using plaintexts of 100 letters (let's assume that these plaintexts agree in only 10 letters and differ in the other 90 letters). Explain why you should win

almost every time. What does this say about ciphertext indistinguishability for Enigma?

Chapter 15 Security Protocols

Up to this point, we have covered many basic cryptographic tools, ranging from encryption algorithms to hash algorithms to digital signatures. A natural question arises: Can we just apply these tools directly to make computers and communications secure?

At first glance, one might think that public key methods are the panacea for all of security. They allow two parties who have never met to exchange messages securely. They also provide an easy way to authenticate the origin of a message and, when combined with hash

functions, these signature operations can be made efficient.

Unfortunately, the answer is definitely no and there are many problems that still remain. In discussing public key algorithms, we never really discussed how the public keys are distributed. We have casually said that Alice will announce her public key for Bob to use. Bob, however, should not be too naive in just believing what he hears. How does he know that it is actually Alice that he is communicating with? Perhaps Alice's evil friend, Mallory, is pretending to be Alice but is actually announcing Mallory's public key instead. Similarly, when you access a website to make a purchase, how do you

know that your transaction is really with a legitimate merchant and that no one has set up a false organization? The real challenge in these problems is the issue of authentication, and Bob should always confirm that he is communicating with Alice before sending any important information.

Combining different cryptographic tools to provide security is much trickier than grabbing algorithms off of the shelf. Instead, security protocols involving the exchange of messages between different entities must be carefully thought out in order to prevent clever attacks. This chapter focuses on such security protocols.

15.1 Intruders-in-the-Middle and Impostors

If you receive an email asking you to go to a website and update your account information, how can you be sure that the website is legitimate? An impostor can easily set up a web page that looks like the correct one but simply records sensitive information and forwards it to Eve. This is an important authentication problem that must be addressed in real-world implementations of cryptographic protocols. One standard solution uses certificates and a trusted authority and will be discussed in Section 15.5. Authentication will also play an important role in the protocols in many

other sections of this chapter.

Another major consideration that must be addressed in communications over public channels is the intruder-in-the-middle attack, which we'll discuss shortly. It is another cause for several of the steps in the protocols we discuss.

15.1.1 Intruder-in-the-Middle Attacks

Eve, who has recently learned the difference between a knight and a rook, claims that she can play two chess grandmasters simultaneously and either win one game or draw both games. The strategy is simple. She waits for the first grandmaster to move, then makes the identical move against the second

grandmaster. When the second grandmaster responds, Eve makes that play against the first grandmaster. Continuing in this way, Eve cannot lose both games (unless she runs into time trouble because of the slight delay in transferring the moves).

A similar strategy, called the **intruder-in-the-middle attack**, can be used against many cryptographic protocols. Many of the technicalities of the algorithms in this chapter are caused by efforts to thwart such an attack.

Let's see how this attack works against the Diffie-Hellman key exchange from Section 10.4.

Let's recall the protocol. Alice and Bob

want to establish a key for communicating. The Diffie-Hellman scheme for accomplishing this is as follows:

1. Either Alice or Bob selects a large, secure prime number p and a primitive root α (mod p). Both p and α can be made public.

2. Alice chooses a secret random x with $1 \leq x \leq p-2$, and Bob selects a secret random y with $1 \leq y \leq p-2$.

3. Alice sends α^x (mod p) to Bob, and Bob sends α^y (mod p) to Alice.

4. Using the messages that they each have received, they can each calculate the session key K. Alice calculates K by $K \equiv (\alpha^y)^x$ (mod p), and Bob calculates K by $K \equiv (\alpha^x)^y$ (mod p).

Here is how the intruder-in-the-middle attack works.

1. Eve chooses an exponent z.

2. Eve intercepts αx and αy.

3. Eve sends αz to Alice and to Bob (Alice believes she is receiving αy and Bob believes he is receiving αx).

4. Eve computes $KAE \equiv (\alpha x)z \pmod{p}$ and $KEB \equiv (\alpha y)z \pmod{p}$. Alice, not realizing that Eve is in the middle, also computes KAE, and Bob computes KEB.

5. When Alice sends a message to Bob, encrypted with KAE, Eve intercepts it, deciphers it, encrypts it with KEB, and sends it to Bob. Bob decrypts with KEB and obtains the message. Bob has no reason to believe the communication was insecure. Meanwhile, Eve is reading the juicy gossip that she has obtained.

To avoid the intruder-in-the-middle attack, it is desirable to have a procedure that authenticates Alice's and Bob's identities to each other while the key is being formed. A protocol that can do this is known as an **authenticated key agreement protocol**.

A standard way to stop the intruder-in-the-middle attack is the **station-to-station (STS) protocol**, which uses digital signatures. Each user U has a digital signature function sigU with verification algorithm verU. For example, sigU could produce an RSA or ElGamal signature, and verU checks that it is a valid signature for U. The verification algorithms are compiled and made public by the trusted authority Trent, who certifies that verU is actually the verification algorithm for U and not for Eve.

Suppose now that Alice and Bob want to establish a key to use in an encryption function EK. They proceed as in the Diffie-Hellman key exchange, but with

the added feature of digital signatures:

1. They choose a large prime p and a primitive root α.
2. Alice chooses a random x and Bob chooses a random y.
3. Alice computes $\alpha^x \pmod p$, and Bob computes $\alpha^y \pmod p$.
4. Alice sends α^x to Bob.
5. Bob computes $K \equiv (\alpha^x)^y \pmod p$.
6. Bob sends α^y and $E_K(\text{sig}_B(\alpha^y, \alpha^x))$ to Alice.
7. Alice computes $K \equiv (\alpha^y)^x \pmod p$.
8. Alice decrypts $E_K(\text{sig}_B(\alpha^y, \alpha^x))$ to obtain $\text{sig}_B(\alpha^y, \alpha^x)$.
9. Alice asks Trent to verify that verB is Bob's verification algorithm.
10. Alice uses verB to verify Bob's signature.
11. Alice sends $E_K(\text{sig}_A(\alpha^x, \alpha^y))$ to Bob.
12. Bob decrypts, asks Trent to verify that verA is Alice's verification algorithm, and then uses verA

to verify Alice's signature.

This protocol is due to Diffie, van Oorschot, and Wiener. Note that Alice and Bob are also certain that they are using the same key K, since it is very unlikely that an incorrect key would give a decryption that is a valid signature.

Note the role that trust plays in the protocol. Alice and Bob must trust Trent's verification if they are to have confidence that their communications are secure. Throughout this chapter, a trusted authority such as Trent will be an important participant in many protocols.

15.2 Key Distribution

So far in this book we have discussed various cryptographic concepts and focused on developing algorithms for secure communication. But a cryptographic algorithm is only as strong as the security of its keys. If Alice were to announce to the whole world her key before starting an AES session with Bob, then anyone could eavesdrop. Such a scenario is absurd, of course. But it represents an extreme version of a very important issue: If Alice and Bob are unable to meet in order to exchange their keys, can they still decide on a key without compromising future communication?

In particular, there is the fundamental problem of sharing secret information for the establishment of keys for symmetric cryptography. By symmetric cryptography, we mean a system such as AES where both the sender and the recipient use the same key. This is in contrast to public key methods such as RSA, where the sender has one key (the encryption exponent) and the receiver has another (the decryption exponent).

In key establishment protocols, there is a sequence of steps that take place between Alice and Bob so that they can share some secret information needed in the establishment of a key. Since public key cryptography methods employ public encryption keys that are stored in

public databases, one might think that public key cryptography provides an easy solution to this problem. This is partially true. The main downside to public key cryptography is that even the best public key cryptosystems are computationally slow when compared with the best symmetric key methods. RSA, for example, requires exponentiation, which is not as fast as the mixing of bits that takes place in AES. Therefore, sometimes RSA is used to transmit an AES key that will then be used for transmitting vast amounts of data. However, a central server that needs to communicate with many clients in short time intervals sometimes needs key establishment methods that are faster than current versions of public

key algorithms. Therefore, in this and in various other situations, we need to consider other means for the exchange and establishment of keys for symmetric encryption algorithms.

There are two basic types of key establishment. In **key agreement** protocols, neither party knows the key in advance; it is determined as a result of their interaction. In **key distribution** protocols, one party has decided on a key and transmits it to the other party.

Diffie-Hellman key exchange (see Sections 10.4 and 15.1) is an example of key agreement. Using RSA to transmit an AES key is an example of key distribution.

In any key establishment protocol, authentication and intruder-in-the-middle attacks are security concerns. Pre-distribution, which will be discussed shortly, is one solution. Another solution involves employing a server that will handle the task of securely giving keys to two entities wishing to communicate. We will also look at some other basic protocols for key distribution using a third party. Solutions that are more practical for Internet communications are treated in later sections of this chapter.

15.2.1 Key Pre-distribution

In the simplest version of this protocol, if Alice wants to communicate with Bob, the keys or key schedules (lists

describing which keys to use at which times) are decided upon in advance and somehow this information is sent securely from one to the other. For example, this method was used by the German navy in World War II. However, the British were able to use codebooks from captured ships to find daily keys and thus read messages.

There are some obvious limitations and drawbacks to pre-distribution. First, it requires two parties, Alice and Bob, to have met or to have established a secure channel between them in the first place. Second, once Alice and Bob have met and exchanged information, there is nothing they can do, other than meeting again, to change the key information in

case it gets compromised. The keys are predetermined and there is no easy method to change the key after a certain amount of time. When using the same key for long periods of time, one runs a risk that the key will become compromised. The more data that are transmitted, the more data there are with which to build statistical attacks.

Here is a general and slightly modified situation. First, we require a trusted authority whom we call Trent. For every pair of users, call them (A, B), Trent produces a random key KAB that will be used as a key for a symmetric encryption method (hence KBA=KAB). It is assumed that Trent is powerful and has established a secure channel to each

of the users. He distributes all the keys that he has determined to his users. Thus, if Trent is responsible for n users, each user will be receiving n−1 keys to store, and Trent must send n(n−1)/2 keys securely. If n is large, this could be a problem. The storage that each user requires is also a problem.

One method for reducing the amount of information that must be sent from the trusted authority is the **Blom key pre-distribution scheme**. Start with a network of n users, and let p be a large prime, where $p \geq n$. Everyone has knowledge of the prime p. The protocol is now the following:

1. Each user U in the network is assigned a distinct public number rU (mod p).

2. Trent chooses three secret random numbers a, b, and c mod p.

3. For each user U, Trent calculates the numbers

   aU ≡ a+brU(mod p)bU ≡ b+crU(mod p)

   and sends them via his secure channel to U.

4. Each user U forms the linear polynomial

   gU(x)=aU+bUx.

5. If Alice (A) wants to communicate with Bob (B), then Alice computes KAB=gArB, while Bob computes KBA=gBrA.

6. It can be shown that KAB=KBA (Exercise 2). Alice and Bob communicate via a symmetric encryption system, for example, AES, using the key (or a key derived from) KAB.

Example

Consider a network consisting of three users Alice, Bob, and Charlie. Let p=23, and let

rA=11, rB=3, rC=2.

Suppose Trent chooses the numbers a=8, b=3, c=1. The corresponding linear polynomials are given by

gA(x)=18+14x, gB(x)=17+6x, gC(x)=14+5x.

It is now possible to calculate the keys that this scheme would generate:

KAB=gA(rB)=14, KAC=gA(rC)=0, KBC=gB(rC)=6.

It is easy to check that KAB=KBA, etc., in this example.

If the two users Eve and Oscar conspire, they can determine a, b, and c and therefore find all numbers aA, bA for all users. They proceed as follows. They know the numbers aE, bE, aO, bO. The defining equations for the last three of these numbers can be written in matrix form as

01rE1rO001rOabc ≡ bEaObO(mod p).

The determinant of the matrix is rE−rO. Since the numbers rA were chosen to be distinct mod p, the determinant is nonzero mod p and therefore the system has a unique solution a, b, c.

Without Eve's help, Oscar has only a 2×3 matrix to work with and therefore cannot find a, b, c. In fact, suppose he wants to calculate the key KAB being used by Alice and Bob. Since KAB ≡ a +b(rA+rB)+c(rArB) (see Exercise 2), there is the matrix equation

1rA+rBrArB1rO001rOabc ≡ KABaObO(mod p).

The matrix has determinant (rO−rA)(rO −rB) ≡ 0 (mod p). Therefore, there is a solution a, b, c for every possible value of KAB. This means that Oscar obtains

no information about KAB.

For each $k \geq 1$, there are Blom schemes that are secure against coalitions of at most k users, but which succumb to conspiracies of k+1 users. See [Blom].

15.2.2 Authenticated Key Distribution

Key pre-distribution schemes are often impractical because they require significant resources to initialize and do not allow for keys to be changed or replaced easily when keys are deemed no longer safe. One way around these problems is to introduce a **trusted authority**, whose task is to distribute new keys to communicating parties as they are needed. This trusted third party

may be a server on a computer network, or an organization that is trusted by both Alice and Bob to distribute keys securely.

Authentication is critical to key distribution. Alice and Bob will ask the trusted third party, Trent, to give them keys. They want to make certain that there are no malicious entities masquerading as Trent and sending them false key messages. Additionally, when Alice and Bob exchange messages with each other, they will each need to make certain that the person they are talking to is precisely the person they think they are talking to.

One of the main challenges facing key distribution is the issue of replay

attacks. In a replay attack, an opponent may record a message and repeat it at a later time in hope of either pretending to be another party or eliciting a particular response from an entity in order to compromise a key. To provide authentication and protect against replay attacks, we need to make certain that vital information, such as keys and identification parameters, are kept confidential. Additionally, we need to guarantee that each message is fresh; that is, it isn't a repeat of a message from a long time ago.

The task of confidentiality can be easily accomplished using existing keys already shared between entities. These keys are used to encrypt messages used

in the distribution of session keys and are therefore often called key encrypting keys. Unfortunately, no matter how we look at it, there is a chicken-and-egg problem: In order to distribute session keys securely, we must assume that entities have already securely shared key encrypting keys with the trusted authority.

To handle message freshness, however, we typically need to attach extra data fields in each message we exchange. There are three main types of data fields that are often introduced in order to prevent replay attacks:

- Sequence numbers: Each message that is sent between two entities has a sequence number associated with it. If an entity ever sees the same sequence number again, then the entity concludes

that the message is a replay. The challenge with sequence numbers is that it requires that each party keep track of the sequence numbers it has witnessed.

- Timestamps: Each message that is sent between two entities has a statement of the time period for when that message is valid. This requires that both entities have clocks that are set to the same time.

- Nonces: A nonce is a random message that is allowed to be used only once and is used as part of a challenge-response mechanism. In a challenge-response, Alice sends Bob a message involving a nonce and requires Bob to send back a correct response to her nonce.

We will now look at two examples of key distribution schemes and analyze attacks that may be used against each in order to bypass the intended security. These two examples should highlight how difficult it is to distribute keys securely.

We begin with a protocol known as the **wide-mouthed frog protocol**, which is one of the simplest symmetric key management protocols involving a trusted authority. In this protocol, Alice chooses a session key KAB to communicate with Bob and has Trent transfer it to Bob securely:

1. Alice→Trent: EKATtA‖IDB‖KAB.

2. Trent→Bob: EKBTtT‖IDA‖KAB.

Here, KAT is a key shared between Alice and Trent, while KBT is a key shared between Bob and Trent. Alice's and Bob's identifying information are given by IDA and IDB, respectively. The parameter tA is a timestamp supplied by Alice, while tT is a timestamp given by Trent. It is assumed

that Alice, Trent, and Bob have synchronized clocks. Bob will accept KAB as fresh if it arrives within a window of time. The key KAB will be valid for a certain period of time after tT.

The purpose behind the two timestamps is to allow Bob to check to see that the message is fresh. In the first message, Alice sends a message with a timestamp tA. If Trent gets the message and the time is not too far off from tA, he will then agree to translate the message and deliver it to Bob.

The problem with the protocol comes from the second message. Here, Trent has updated the timestamp to a newer timestamp tT. Unfortunately, this simple

change allows for a clever attack in which the nefarious Mallory may cause Trent to extend the lifetime of an old key. Let us step through this attack.

1. After seeing one exchange of the protocol, Mallory pretends to be Bob wanting to share a key with Alice. Mallory sends Trent the replay EKBTtT‖IDA‖KAB.

2. Trent sends EKATtT ′‖IDB‖KAB to Alice, with a new timestamp tT ′. Alice thinks this is a valid message since it came from Trent and was encrypted using Trent's key. The key KAB will now be valid for a period of time after tT ′.

3. Mallory then pretends to be Alice and gets EKBTtT ″‖IDA‖KAB. The key KAB will now be valid for a period of time after tT″>tT′.

4. Mallory continues alternately playing Trent against Bob and then Trent against Alice.

The net result is that the malicious Mallory can use Trent as an agent to force Alice and Bob to continue to use

KAB indefinitely. Of course, Alice and Bob should keep track of the fact that they have seen KAB before and begin to suspect that something suspicious is going on when they repeatedly see KAB. The protocol did not explicitly state that this was necessary, however, and security protocols should be very explicit on what it is that they assume and don't assume. The true problem, though, is the fact that Trent replaces tA with tT. If Trent had not changed tT and instead had left tA as the timestamp, then the protocol would have been better off.

Another example of an authenticated key exchange protocol is due to Needham and Schroeder. In the

Needham–Schroeder protocol, Alice and Bob wish to obtain a session key KS from Trent so that they can talk with each other. The protocol involves the following steps:

1. Alice→Trent: IDA‖IDB‖r1
2. Trent→Alice: EKATKS‖IDB‖r1‖EKBT[KS‖IDA]
3. Alice→Bob: EKBTKS‖IDA
4. Bob→Alice: EKS[r2]
5. Alice→Bob: EKS[r2−1]

Just as in the earlier protocol, KAT is a key shared between Alice and Trent, while KBT is a key shared between Bob and Trent. Unlike the wide-mouthed frog protocol, the Needham–Schroeder protocol does not employ timestamps but instead uses random numbers r1 and

r2 as nonces. In the first step, Alice sends Trent her request, which is a statement of who she is and whom she wants to talk to, along with a random number r1. Trent gives Alice the session key KS and gives Alice a package EKBT[KS‖IDA] that she will deliver to Bob. In the next step, she delivers the package to Bob. Bob can decrypt this to get the session key and the identity of the person he is talking with. Next, Bob sends Alice his own challenge by sending the second nonce r2. In the final step, Alice proves her identity to Bob by answering his challenge. Using r2−1 instead of r2 prevents Mallory from replaying message 4.

The purpose of r1 is to prevent the reuse

of old messages. Suppose r1 is omitted from Steps 1 and 2. If Eve sees that Alice wants to communicate with Bob, she could intercept Trent's message in Step 2 and substitute a transmission from a previous Step 2. Then Alice and Bob would communicate with a previously used session key, something that should generally be avoided. When r1 is omitted, Alice has no way of knowing that this happened unless she has kept a list of previous session keys KS.

Observe that the key exchange portion of the protocol is completed at the end of the third step. The last two exchanges, however, seem a little out of place and deserve some more

discussion. The purpose of the nonce in step 4 and step 5 is to prevent replay attacks in which Mallory sends an old EKBT[KS‖IDA] to Bob. If we didn't have step 4 and step 5, Bob would automatically assume that KS is the correct key to use. Mallory could use this strategy to force Bob to send out more messages to Alice involving KS. Step 4 and step 5 allow Bob to issue a challenge to Alice where she can prove to Bob that she really knows the session key KS. Only Alice should be able to use KS to calculate EKS[r2−1].

In spite of the the apparent security that the challenge-response in step 4 and step 5 provides, there is a potential security problem that can arise if Mallory ever

figures out the session key KS. Let us step through this possible attack.

1. Mallory→Bob: EKBT [KS‖IDA]
2. Bob→Alice: EKS [r3]
3. Mallory→Bob: EKS [r3−1]

Here, Mallory replays an old message from step 3 of Needham–Schroeder as if Mallory were Alice. When Bob gets this message, he issues a challenge to Alice in the form of a new nonce r3. Mallory can intercept this challenge and, since she knows the session key KS, she can respond correctly to the challenge. The net result is that Mallory will have passed Bob's authentication challenge as if she were Alice. From this point on, Bob will communicate using KS and believe he is communicating with Alice.

Mallory can use Alice's identity to complete her evil plans.

Building a solid key distribution protocol is very tough. There are many examples in the security literature of key distribution schemes that have failed because of a clever attack that was found years later. It might seem a lost cause since we have examined two protocols that both have weaknesses associated with them. However, in the rest of this chapter we shall look at protocols that have so far proven successful. We begin our discussion of successful protocols in the next section, where we will discuss Kerberos, which is an improved variation of the Needham–Schroeder key exchange

protocol. Kerberos has withstood careful scrutiny by the community and has been adopted for use in many applications.

15.3 Kerberos

Kerberos (named for the three-headed dog that guarded the entrance to Hades) is a real-world implementation of a symmetric cryptography protocol whose purpose is to provide strong levels of authentication and security in key exchange between users in a network. Here we use the term *users* loosely, as a user might be an individual, or it might be a program requesting communication with another program. Kerberos grew out of a larger development project at MIT known as Project Athena. The purpose of Athena was to provide a huge network of computer workstations for the undergraduate student body at

MIT, allowing students to access their files easily from anywhere on the network. As one might guess, such a development quickly raised questions about network security. In particular, communication across a public network such as Athena is very insecure. It is easily possible to observe data flowing across a network and look for interesting bits of information such as passwords and other types of information that one would wish to remain private. Kerberos was developed in order to address such security issues. In the following, we present the basic Kerberos model and describe what it is and what it attempts to do. For more thorough descriptions, see [Schneier].

Kerberos is based on a client/server architecture. A client is either a user or some software that has some task that it seeks to accomplish. For example, a client might wish to send email, print documents, or mount devices. Servers are larger entities whose function is to provide services to the clients. As an example, on the Internet and World Wide Web there is a concept of a domain name server (DNS), which provides names or addresses to clients such as email programs or Internet browsers.

The basic Kerberos model has the following participants:

- Cliff: a client
- Serge: a server

- Trent: a trusted authority
- Grant: a ticket-granting server

The trusted authority is also known as an authentication server. To begin, Cliff and Serge have no secret key information shared between them, and it is the purpose of Kerberos to give each of them information securely. A result of the Kerberos protocol is that Serge will have verified Cliff's identity (he wouldn't want to have a conversation with a fake Cliff, would he?), and a session key will be established.

The protocol, depicted in Figure 15.1, begins with Cliff requesting a ticket for ticket-granting service from Trent. Since Trent is the powerful trusted authority, he has a database of password

information for all the clients (for this reason, Trent is also sometimes referred to as the Kerberos server). Trent returns a ticket that is encrypted with the client's secret password information. Cliff would now like to use the service that Serge provides, but before he can do this, he must be allowed to talk to Serge. Cliff presents his ticket to Grant, the ticket-granting server. Grant takes this ticket, and if everything is OK (recall that the ticket has some information identifying Cliff), then Grant gives a new ticket to Cliff that will allow Cliff to make use of Serge's service (and only Serge's service; this ticket will not be valid with Sarah, a different server). Cliff now has a service ticket, which he can present to Serge.

He sends Serge the service ticket as well as an authentication credential. Serge checks the ticket with the authentication credential to make sure it is valid. If this final exchange checks out, then Serge will provide the service to Cliff.

Figure 15.1 Kerberos.

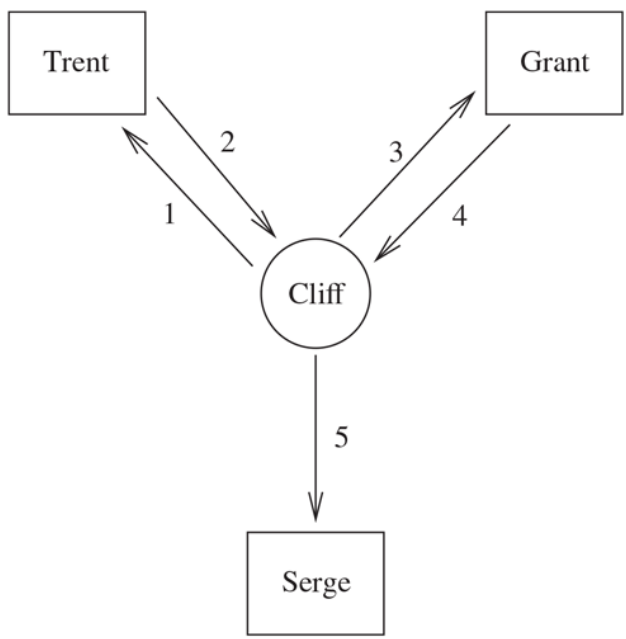


Figure 15.1 Full Alternative Text

The Kerberos protocol is a formal

version of protocols we use in everyday life, where different entities are involved in authorizing different steps in a process; for example, using an ATM to get cash, then buying a ticket for a ride at a fair.

We now look at Kerberos in more detail. Kerberos makes use of a symmetric encryption algorithm. In the original release of Kerberos Version V, Kerberos used DES operating in CBC mode; however, Version V was later updated to allow for more general symmetric encryption algorithms.

1. Cliff to Trent: Cliff sends a message to Trent that contains his name and the name of the ticket-granting server that he will use (in this case Grant).

2. Trent to Cliff: Trent looks up Cliff's name in his database. If he finds it, he generates a session key

KCG that will be used between Cliff and Grant. Trent also has a secret key KC with which he can communicate with Cliff, so he uses this to encrypt the Cliff–Grant session key:

T=eKC(KCG).

In addition, Trent creates a Ticket Granting Ticket (TGT), which will allow Cliff to authenticate himself to Grant. This ticket is encrypted using Grant's personal key KG (which Trent also has):

TGT=Grant's name‖eKG(Cliff's name, Cliff's Address, Timestamp1, KCG).

Here ‖ is used to denote concatenation. The ticket that Cliff receives is the concatenation of these two subtickets:

Ticket =T‖TGT.

3. Cliff to Grant: Cliff can extract KCG using the key KC, which he shares with Trent. Using KCG, Cliff can now communicate securely with Grant. Cliff now creates an authenticator, which will consist of his name, his address, and a timestamp. He encrypts this using KCG to get

   AuthCG = eKCG(Cliff's name, Cliff's address, Timestamp2).

   Cliff now sends AuthCG as well as TGT to Grant

so that Grant can administer a service ticket.

4. Grant to Cliff: Grant now has AuthCG and TGT. Part of TGT was encrypted using Grant's secret key, so Grant can extract this portion and can decrypt it. Thus he can recover Cliff's name, Cliff's address, Timestamp1, as well as KCG. Grant can now use KCG to decrypt AuthCG in order to verify the authenticity of Cliff's request. That is, dKCGAuthCG will provide another copy of Cliff's name, Cliff's address, and a different timestamp. If the two versions of Cliff's name and address match, and if Timestamp1 and Timestamp2 are sufficiently close to each other, then Grant will declare Cliff valid. Now that Cliff is approved by Grant, Grant will generate a session key KCS for Cliff to communicate with Serge and will also return a service ticket. Grant has a secret key KS, which he shares with Serge. The service ticket is

   ServTicket =eKSCliff's name, Cliff's address, Timestamp3, ExpirationTime, KCS.

   Here ExpirationTime is a quantity that describes the length of validity for this service ticket. The session key is encrypted using a session key between Cliff and Grant:

   eKCGKCS.

Grant sends ServTicket and eKCGKCS to Cliff.

5. Cliff to Serge: Cliff is now ready to start making use of Serge's services. He starts by decrypting eKCGKCS in order to get the session key KCS that he will use while communicating with Serge. He creates an authenticator to use with Serge:

   AuthCS =eKCSCliff's name, Cliff's address, Timestamp4.

   Cliff now sends Serge AuthCS as well as ServTicket. Serge can decrypt ServTicket and extract from this the session key KCS that he is to use with Cliff. Using this session key, he can decrypt AuthCS and verify that Cliff is who he says he is, and that Timestamp4 is within ExpirationTime of Timestamp3. If Timestamp4 is not within ExpirationTime of Timestamp3, then Cliff's ticket is stale and Serge rejects his request for service. Otherwise, Cliff and Serge may make use of KCS to perform their exchange.

Some comments about the different versions of Kerberos is appropriate. There are two main versions of Kerberos that one will find discussed in

the literature: Kerberos Version IV, which was developed originally as part of MIT's Project Athena; and Kerberos Version V, which was originally published in 1993 and was intended to address limitations and weaknesses in Version IV. Kerberos Version V was subsequently revised in 2005. We now describe some of the differences between the two versions.

Both Version IV and Version V follow the basic model that we have presented. Kerberos Version IV was designed to work with the computer networks associated with Project Athena, and consequently Version V was enhanced to support authentication and key exchange on larger networks. In

particular, Kerberos Version IV was limited in its use to Internet Protocol version 4 (IPV4) addresses to specify clients and servers, while Kerberos Version V expanded the use of network protocols to support multiple IP addresses and addresses associated with other types of network protocols, such as the longer addresses of Internet Protocol version 6 (IPV6).

In terms of the cryptography that Kerberos uses, Version IV only allowed the use of the Data Encryption Standard (DES) with a nonstandard mode of operation known as Propagating Cipher Block Chaining (PCBC). Due to concerns about the security of PCBC mode, it was removed in Version V.

Instead, Version V allows for the use of more general symmetric encryption algorithms, such as the use of AES, and incorporated integrity checking mechanisms directly into its use of the normal CBC mode of operation.

The tickets used in Kerberos Version IV had strict limits to their duration. In Kerberos Version V, the representation for the tickets was expanded to include start and stop times for the certificates. This allows for tickets in Version V to be specified with arbitrary durations. Additionally, the functionality of tickets in Version V was expanded. In Version V, it is possible for servers to forward, renew, and postdate tickets, while in Version IV authentication forwarding is

not allowed. Overall, the changes made to Version V were intended to make Kerberos more secure and more flexible, allowing it to operate in more types of networks, support a broader array of cipher algorithms, and allow for it to address a broader set of application requirements.

15.4 Public Key Infrastructures (PKI)

Public key cryptography is a powerful tool that allows for authentication, key distribution, and non-repudiation. In these applications, the public key is published, but when you access public keys, what assurance do you have that Alice's public key actually belongs to Alice? Perhaps Eve has substituted her own public key in place of Alice's. Unless confidence exists in how the keys were generated, and in their authenticity and validity, the benefits of public key cryptography are minimal.

In order for public key cryptography to

be useful in commercial applications, it is necessary to have an infrastructure that keeps track of public keys. A public key infrastructure, or PKI for short, is a framework consisting of policies defining the rules under which the cryptographic systems operate and procedures for generating and publishing keys and certificates.

All PKIs consist of certification and validation operations. Certification binds a public key to an entity, such as a user or a piece of information. Validation guarantees that certificates are valid.

A **certificate** is a quantity of information that has been signed by its publisher, who is commonly referred to as the **certification authority (CA)**.

There are many types of certificates. Two popular ones are identity certificates and credential certificates. Identity certificates contain an entity's identity information, such as an email address, and a list of public keys for the entity. Credential certificates contain information describing access rights. In either case, the data are typically encrypted using the CA's private key.

Suppose we have a PKI, and the CA publishes identity certificates for Alice and Bob. If Alice knows the CA's public key, then she can take the encrypted identity certificate for Bob that has been published and extract Bob's identity information as well as a list of public keys needed to

communicate securely with Bob. The difference between this scenario and the conventional public key scenario is that Bob doesn't publish his keys, but instead the trust relationship is placed between Alice and the publisher. Alice might not trust Bob as much as she might trust a CA such as the government or the phone company. The concept of trust is critical to PKIs and is perhaps one of the most important properties of a PKI.

It is unlikely that a single entity could ever keep track of and issue every Internet user's public keys. Instead, PKIs often consist of multiple CAs that are allowed to certify each other and the certificates they issue. Thus, Bob might

be associated with a different CA than Alice, and when requesting Bob's identity certificate, Alice might only trust it if her CA trusts Bob's CA. On large networks like the Internet, there may be many CAs between Alice and Bob, and it becomes necessary for each of the CAs between her and Bob to trust each other.

In addition, most PKIs have varying levels of trust, allowing some CAs to certify other CAs with varying degrees of trust. It is possible that CAs may only trust other CAs to perform specific tasks. For example, Alice's CA may only trust Bob's CA to certify Bob and not certify other CAs, while Alice's CA may trust Dave's CA to certify other

CAs. Trust relationships can become very elaborate, and, as these relationships become more complex, it becomes more difficult to determine to what degree Alice will trust a certificate that she receives.

In the following two sections, we discuss two examples of PKIs that are used in practice.

15.5 X.509 Certificates

Suppose you want to buy something on the Internet. You go to the website Gigafirm.com, select your items, and then proceed to the checkout page. You are asked to enter your credit card number and other information. The website assures you that it is using secure public key encryption, using Gigafirm's public key, to set up the communications. But how do you know that Eve hasn't substituted her public key? In other words, when you are using public keys, how can you be sure that they are correct? This is the purpose of Digital Certificates.

One of the most popular types of certificate is the X.509. In this system, every user has a certificate. The validity of the certificates depends on a chain of trust. At the top is a **certification authority** (CA). These are often commercial companies such as VeriSign, GTE, AT&T, and others. It is assumed that the CA is trustworthy. The CA produces its own certificate and signs it. This certificate is often posted on the CA's website. In order to ensure that their services are used frequently, various CAs arrange to have their certificates packaged into Internet browsers such as Chrome, Firefox, Safari, Internet Explorer, and Edge.

The CA then (for a fee) produces

certificates for various clients, such as Gigafirm. Such a certificate contains Gigafirm's public key. It is signed by the CA using the CA's private key. Often, for efficiency, the CA authorizes various **registration authorities** (RA) to sign certificates. Each RA then has a certificate signed by the CA.

A certificate holder can sometimes then sign certificates for others. We therefore get a **certification hierarchy** where the validity of each certificate is certified by the user above it, and this continues all the way up to the CA.

If Alice wants to verify that Gigafirm's public key is correct, she uses her copy of the CA's certificate (stored in her computer) to get the CA's public key.

She then uses it to verify the signature on Gigafirm's certificate. If it is valid, she trusts the certificate and thus has a trusted public key for Gigafirm. Of course, she must trust the CA's public key. This means that she trusts the company that packaged the CA's certificate into her computer. The computer company of course has a financial incentive to maintain a good reputation, so this trust is reasonable. But if Alice has bought a used computer in which Eve has tampered with the certificates, there might be a problem (in other words, don't buy used computers from your enemies, except to extract unerased information).

Figure 15.2 A Certification

Hierarchy.

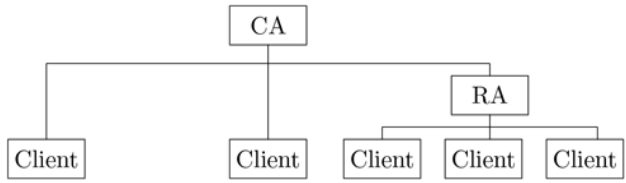


Figure 15.2 Full Alternative Text

Figures 15.3, 15.4, and 15.5 show examples of X.509 certificates. The ones in Figures 15.3 and 15.4 are for a CA, namely VeriSign. The part in Figure 15.3 gives the general information about the certificate, including its possible uses. Figure 15.4 gives the detailed information. The one in Figure 15.5 is an edited version of the Details part of a certificate for the bank Wells Fargo.

Figure 15.3 CA's Certificate; General.

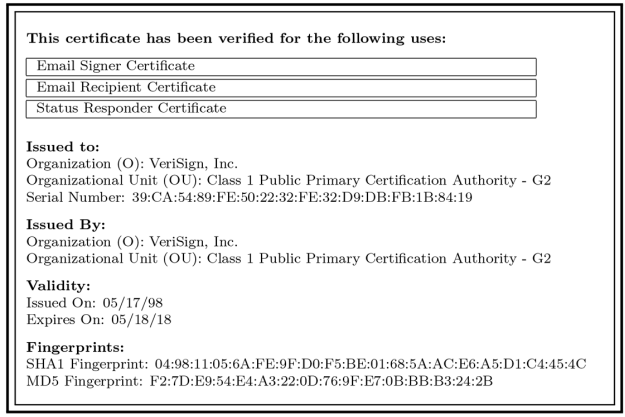

This certificate has been verified for the following uses:
Email Signer Certificate
Email Recipient Certificate
Status Responder Certificate
Issued to:
Organization (O): VeriSign, Inc.
Organizational Unit (OU): Class 1 Public Primary Certification Authority - G2
Serial Number: 39:CA:54:89:FE:50:22:32:FE:32:D9:DB:FB:1B:84:19
Issued By:
Organization (O): VeriSign, Inc.
Organizational Unit (OU): Class 1 Public Primary Certification Authority - G2
Validity:
Issued On: 05/17/98
Expires On: 05/18/18
Fingerprints:
SHA1 Fingerprint: 04:98:11:05:6A:FE:9F:D0:F5:BE:01:68:5A:AC:E6:A5:D1:C4:45:4C
MD5 Fingerprint: F2:7D:E9:54:E4:A3:22:0D:76:9F:E7:0B:BB:B3:24:2B

Figure 15.3 Full Alternative Text

Figure 15.4 CA's Certificate; Details.

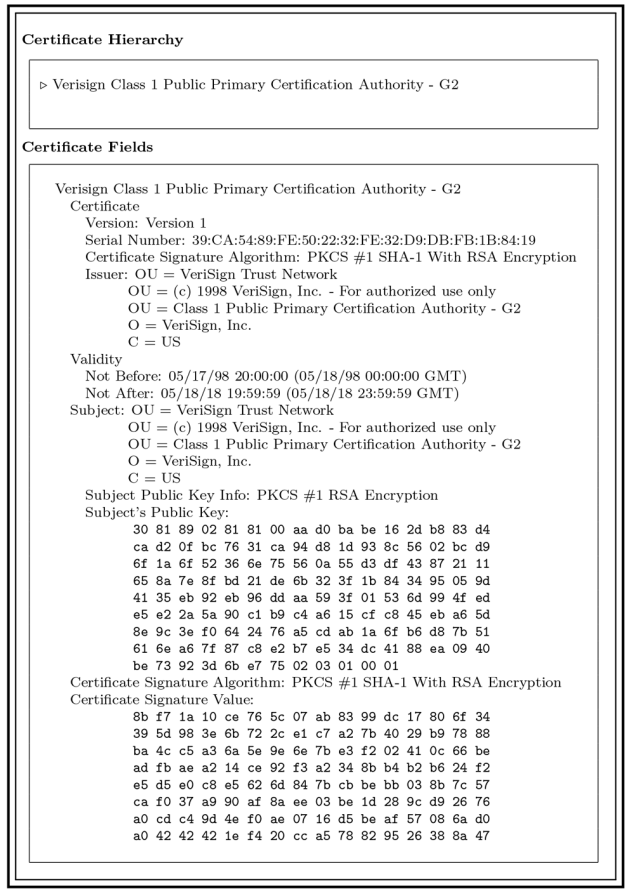

Certificate Hierarchy
▹ Verisign Class 1 Public Primary Certification Authority - G2
Certificate Fields
Verisign Class 1 Public Primary Certification Authority - G2
Certificate
Version: Version 1
Serial Number: 39:CA:54:89:FE:50:22:32:FE:32:D9:DB:FB:1B:84:19
Certificate Signature Algorithm: PKCS #1 SHA-1 With RSA Encryption
Issuer: OU = VeriSign Trust Network
OU = (c) 1998 VeriSign, Inc. - For authorized use only
OU = Class 1 Public Primary Certification Authority - G2
O = VeriSign, Inc.
C = US
Validity
Not Before: 05/17/98 20:00:00 (05/18/98 00:00:00 GMT)
Not After: 05/18/18 19:59:59 (05/18/18 23:59:59 GMT)
Subject: OU = VeriSign Trust Network
OU = (c) 1998 VeriSign, Inc. - For authorized use only
OU = Class 1 Public Primary Certification Authority - G2
O = VeriSign, Inc.
C = US
Subject Public Key Info: PKCS #1 RSA Encryption
Subject's Public Key:
30 81 89 02 81 81 00 aa d0 ba be 16 2d b8 83 d4
ca d2 0f bc 76 31 ca 94 d8 1d 93 8c 56 02 bc d9
6f 1a 6f 52 36 6e 75 56 0a 55 d3 df 43 87 21 11
65 8a 7e 8f bd 21 de 6b 32 3f 1b 84 34 95 05 9d
41 35 eb 92 eb 96 dd aa 59 3f 01 53 6d 99 4f ed
e5 e2 2a 5a 90 c1 b9 c4 a6 15 cf c8 45 eb a6 5d
8e 9c 3e f0 64 24 76 a5 cd ab 1a 6f b6 d8 7b 51
61 6e a6 7f 87 c8 e2 b7 e5 34 dc 41 88 ea 09 40
be 73 92 3d 6b e7 75 02 03 01 00 01
Certificate Signature Algorithm: PKCS #1 SHA-1 With RSA Encryption
Certificate Signature Value:
8b f7 1a 10 ce 76 5c 07 ab 83 99 dc 17 80 6f 34
39 5d 98 3e 6b 72 2c e1 c7 a2 7b 40 29 b9 78 88
ba 4c c5 a3 6a 5e 9e 6e 7b e3 f2 02 41 0c 66 be
ad fb ae a2 14 ce 92 f3 a2 34 8b b4 b2 b6 24 f2
e5 d5 e0 c8 e5 62 6d 84 7b cb be bb 03 8b 7c 57
ca f0 37 a9 90 af 8a ee 03 be 1d 28 9c d9 26 76
a0 cd c4 9d 4e f0 ae 07 16 d5 be af 57 08 6a d0
a0 42 42 42 1e f4 20 cc a5 78 82 95 26 38 8a 47

Figure 15.4 Full Alternative Text

Figure 15.5 A Client's Certificate.

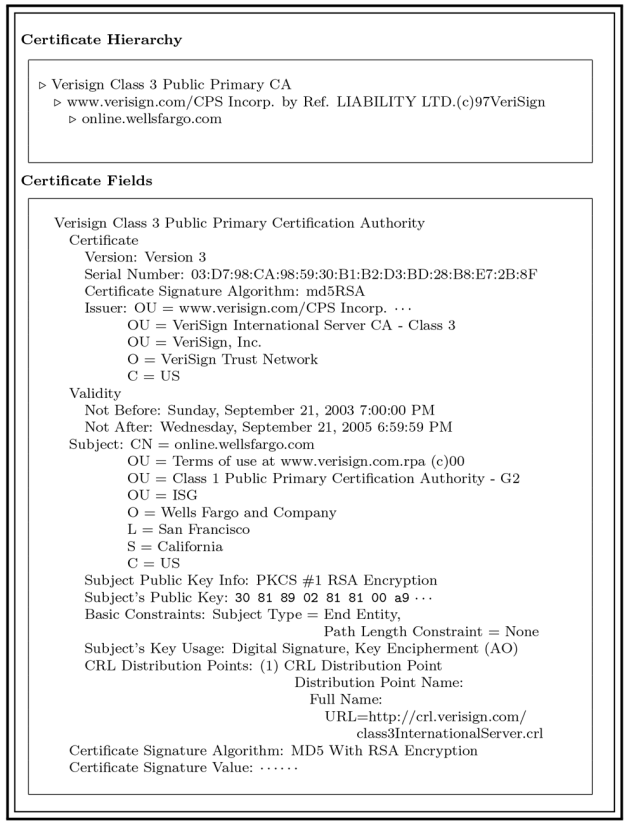

Certificate Hierarchy
▹ Verisign Class 3 Public Primary CA
▹ www.verisign.com/CPS Incorp. by Ref. LIABILITY LTD.(c)97VeriSign
▹ online.wellsfargo.com
Certificate Fields
Verisign Class 3 Public Primary Certification Authority
Certificate
Version: Version 3
Serial Number: 03:D7:98:CA:98:59:30:B1:B2:D3:BD:28:B8:E7:2B:8F
Certificate Signature Algorithm: md5RSA
Issuer: OU = www.verisign.com/CPS Incorp. ···
OU = VeriSign International Server CA - Class 3
OU = VeriSign, Inc.
O = VeriSign Trust Network
C = US
Validity
Not Before: Sunday, September 21, 2003 7:00:00 PM
Not After: Wednesday, September 21, 2005 6:59:59 PM
Subject: CN = online.wellsfargo.com
OU = Terms of use at www.verisign.com.rpa (c)00
OU = Class 1 Public Primary Certification Authority - G2
OU = ISG
O = Wells Fargo and Company
L = San Francisco
S = California
C = US
Subject Public Key Info: PKCS #1 RSA Encryption
Subject's Public Key: 30 81 89 02 81 81 00 a9 ···
Basic Constraints: Subject Type = End Entity,
Path Length Constraint = None
Subject's Key Usage: Digital Signature, Key Encipherment (AO)
CRL Distribution Points: (1) CRL Distribution Point
Distribution Point Name:
Full Name:
URL=http://crl.verisign.com/
class3InternationalServer.crl
Certificate Signature Algorithm: MD5 With RSA Encryption
Certificate Signature Value: ·····

Figure 15.5 Full Alternative Text

Some of the fields in Figure 15.4 are as follows:

1. Version: There are three versions, the first being *Version 1* (from 1988) and the most recent being *Version 3* (from 1997).

2. Serial number: There is a unique serial number for each certificate issued by the CA.

3. Signature algorithm: Various signature algorithms can be used. This one uses *RSA* to sign the output of the hash function *SHA-1*.

4. Issuer: The name of the CA that created and signed this certificate. *OU* is the organizational unit, O is the organization, C is the country.

5. Subject: The name of the holder of this certificate.

6. Public key: Several options are possible. This one uses RSA with a 1024-bit modulus. The key is given in hexadecimal notation. In hexadecimal, the letters a, b, c, d, e, f represent the numbers *10, 11, 12, 13, 14, 15*. Each pair of symbols is a byte, which is eight bits. For example, *b6* represents *11, 6*, which is *10110110* in binary.

   The last three bytes of the public key are `01 00 01`, which is 65537=216+1. This is a very common encryption exponent e for RSA, since raising something to this power by successive squaring (see Section 3.5) is fast. The preceding bytes `02 03` and the bytes `30 81 89 02 81 81 00` at the beginning of the key are control symbols. The remaining 128 bytes `aa d0 ba ⋯ 6b e7 75` are the 1024-bit RSA modulus n.

7. Signature: The preceding information on the certificate is hashed using the hash algorithm

specified – in this case, *SHA-1* – and then signed by raising to the CA's private RSA decryption exponent.

The certificate in Figure 15.5 has a few extra lines. One notable entry is under the heading *Certificate Hierarchy*. The certificate of Wells Fargo has been signed by the Registration Authority (RA) on the preceding line. In turn, the RA's certificate has been signed by the root CA. Another entry worth noting is *CRL Distribution Points.* This is the **certificate revocation list**. It contains lists of certificates that have been revoked. There are two common methods of distributing the information from these lists to the users. Neither is perfect. One way is to send out announcements whenever a certificate is

revoked. This has the disadvantage of sending a lot of irrelevant information to most users (most people don't need to know if the Point Barrow Sunbathing Club loses its certificate). The second method is to maintain a list (such as the one at the listed URL) that can be accessed whenever needed. The disadvantage here is the delay caused by checking each certificate. Also, such a website could get overcrowded if many people try to access it at once. For example, if everyone tries to trade stocks during their lunch hour, and the computers check each certificate for revocation during each transaction, then a site could be overwhelmed.

When Alice (or, usually, her computer)

wants to check the validity of the certificate in Figure 15.5, she sees from the certificate hierarchy that VeriSign's RA signed Wells Fargo's certificate and the RA's certificate was signed by the root CA. She verifies the signature on Wells Fargo's certificate by using the public key (that is, the RSA pair (n, e)) from the RA's certificate; namely, she raises the encrypted hash value to the eth power mod n. If this equals the hash of Wells Fargo's certificate, then she trusts Wells Fargo's certificate, as long as she trusts the RA's certificate. Similarly, she can check the RA's certificate using the public key on the root CA's certificate. Since she received the root CA's certificate from a reliable source (for example, it was packaged in

her Internet browser, and the company doing this has a financial incentive to keep a good reputation), she trusts it. In this way, Alice has established the validity of Wells Fargo's certificate. Therefore, she can confidently do online transactions with Wells Fargo.

There are two levels of certificates. The **high assurance** certificates are issued by the CA under fairly strict controls. High assurance certificates are typically issued to commercial firms. The **low assurance** certificates are issued more freely and certify that the communications are from a particular source. Therefore, if Bob obtains such a certificate for his computer, the certificate verifies that it is Bob's

computer but does not tell whether it is Bob or Eve using the computer. The certificates on many personal computers contain the following line:

Subject: Verisign Class 1 CA Individual Subscriber - Persona Not Validated.

This indicates that the certificate is a low assurance certificate. It does not make any claim as to the identity of the user.

If your computer has Edge, for example, click on *Tools*, then *Internet Options*, then *Content*, then *Certificates*. This will allow you to find the CA's whose certificates have been packaged with the browser. Usually, the validity of most of them has not been checked. But for the

accepted ones, it is possible to look at the **certification path**, which gives the path (often one step) from the user's computer's certificate back to the CA.

15.6 Pretty Good Privacy

Pretty Good Privacy, more commonly known as *PGP*, was developed by Phil Zimmerman in the late 1980s and early 1990s. In contrast to X.509 certificates, PGP is a very decentralized system with no CA. Each user has a certificate, but the trust in this certificate is certified to various degrees by other users. This creates a **web of trust**.

For example, if Alice knows Bob and can verify directly that his certificate is valid, then she signs his certificate with her public key. Charles trusts Alice and has her public key, and therefore can check that Alice's signature on Bob's

certificate is valid. Charles then trusts Bob's certificate. However, this does not mean that Charles trusts certificates that Bob signs – he trusts Bob's public key. Bob could be gullible and sign every certificate that he encounters. His signature would be valid, but that does not mean that the certificate is.

Alice maintains a file with a **keyring** containing the trust levels Alice has in various people's signatures. There are varying levels of trust that she can assign: no information, no trust, partial trust, and complete trust. When a certificate's validity is being judged, the PGP program accepts certificates that are signed by someone Alice trusts, or a sufficient combination of partial trusts.

Otherwise it alerts Alice and she needs to make a choice on whether to proceed.

The primary use of PGP is for authenticating and encrypting email. Suppose Alice receives an email asking for her bank account number so that Charles can transfer millions of dollars into her account. Alice wants to be sure that this email comes from Charles and not from Eve, who wants to use the account number to empty Alice's account. In the unlikely case that this email actually comes from her trusted friend Charles, Alice sends her account information, but she should encrypt it so that Eve cannot intercept it and empty Alice's account. Therefore, the first email needs authentication that proves

that it comes from Charles, while the second needs encryption. There are also cases where both authentication and encryption are desirable. We'll show how PGP handles these situations.

To keep the discussion consistent, we'll always assume that Alice is sending a message to Bob. Alice's RSA public key is (n, e) and her private key is d.

Authentication.

1. Alice uses a hash function and computes the hash of the message.

2. Alice signs the hash by raising it to her secret decryption exponent d mod n. The resulting hash code is put at the beginning of the message, which is sent to Bob.

3. Bob raises the hash code to Alice's public RSA exponent e. The result is compared to the hash of the rest of the message.

4. If the result agrees with the hash, and if Bob trusts Alice's public key, the message is accepted as coming from Alice.

   This authentication is the RSA signature method from Section 13.1. Note the role that trust plays. If Bob does not trust Alice's public key as belonging to Alice, then he cannot be sure that the message did not come from Eve, with Eve's signature in place of Alice's.

Encryption.

1. Alice's computer generates a random number, usually 128 bits, to be used as the session key for a symmetric private key encryption algorithm such as *3DES, IDEA*, or *CAST-128* (these last two are block ciphers using 128-bit keys).

2. Alice uses the symmetric algorithm with this session key to encrypt her message.

3. Alice encrypts the session key using Bob's public key.

4. The encrypted key and the encrypted message are sent to Bob.

5. Bob uses his private RSA key to decrypt the

session key. He then uses the session key to decrypt Alice's message.

The combination of a public key algorithm and a symmetric algorithm is used because encryption is generally faster with symmetric algorithms than with public key algorithms. Therefore, the public key algorithm RSA is used for the small encryption of the session key, and then the symmetric algorithm is used to encrypt the potentially much larger message.

Authentication and Encryption

1. Alice hashes her message and signs the hash to obtain the hash code, as in step (2) of the authentication procedure described previously. This hash code is put at the beginning of the message.

2. Alice produces a random 128-bit session key and uses a symmetric algorithm with this session key to encrypt the hash code together with the message, as in the encryption procedure described previously.

3. Alice uses Bob's public key to encrypt the session key.

4. The encrypted session key and the encryption of the hash code and message are sent to Bob.

5. Bob uses his private key to decrypt the session key.

6. Bob uses the session key to obtain the hash code and message.

7. Bob verifies the signature by using Alice's public key, as in the authentication procedure described previously.

   Of course, this procedure requires that Bob trusts Alice's public key certificate. Also, the reason the signature is done before the encryption is so that Bob can discard the session key after decrypting and therefore store the plaintext message with its signature.

To set up a PGP certificate, Alice's computer uses random input obtained from keystrokes, timing, mouse movements, etc. to find primes p, q and then produce an RSA modulus n=pq and encryption and decryption exponents e

and d. The numbers n and e are then Alice's public key. Alice also chooses a secret passphrase. The secret key d is stored securely in her computer. When the computer needs to use her private key, the computer asks her for her passphrase to be sure that Alice is the correct person. This prevents Eve from using Alice's computer and pretending to be Alice. The advantage of the passphrase is that Alice is not required to memorize or type in the decryption exponent d, which is probably more than one hundred digits long.

In the preceding, we have used RSA for signatures and for encryption of the session keys. Other possibilities are allowed. For example, Diffie-Hellman

can be used to establish the session key, and DSA can be used to sign the message.

15.7 SSL and TLS

If you have ever paid for anything over the Internet, your transactions were probably kept secret by SSL or its close relative TLS. Secure Sockets Layer (SSL) was developed by Netscape in order to perform http communications securely. The first version was released in 1994. Version 3 was released in 1995. Transport Layer Security (TLS) is a slight modification of SSL version 3 and was released by the Internet Engineering Task Force in 1999. These protocols are designed for communications between computers with no previous knowledge of each other's capabilities.

In the following, we'll describe SSL version 3. TLS differs in a few minor details such as how the pseudorandom numbers are calculated. SSL consists of two main components. The first component is known as the record protocol and is responsible for compressing and encrypting the bulk of the data sent between two entities. The second component is a collection of management protocols that are responsible for setting up and maintaining the parameters used by the record protocol. The main part of this component is called the handshake protocol.

We will begin by looking at the handshake protocol, which is the most

complicated part of SSL. Let us suppose that Alice has bought something online from Gigafirm and wants to pay for her purchase. The handshake protocol performs authentication between Alice's computer and the server at Gigafirm and is used to allow Alice and Gigafirm to agree upon various cryptographic algorithms. Alice's computer starts by sending Gigafirm's computer a message containing the following:

1. The highest version of SSL that Alice's computer can support

2. A random number consisting of a 4-byte timestamp and a 28-byte random number

3. A Cipher Suite containing, in decreasing order of preference, the algorithms that Alice's computer wants to use for public key (for example, RSA, Diffie-Hellman, ...), block cipher encryption (3DES, DES, AES, ...), hashing (SHA-1, MD5, ...), and compression (PKZip, ...)

Gigafirm's computer responds with a random 32-byte number (chosen similarly) and its choices of which algorithms to use; for example, RSA, DES, SHA-1, PKZip.

Gigafirm's computer then sends its X.509 certificate (and the certificates in its certification chain). Gigafirm can ask for Alice's certificate, but this is rarely done for two reasons. First, it would impede the transaction, especially if Alice does not have a valid certificate. This would not help Gigafirm accomplish its goal of making sales. Secondly, Alice is going to send her credit card number later in the transaction, and this serves to verify that Alice (or the thief who picked her

pocket) has Alice's card.

We'll assume from now on that RSA was chosen for the public key method. The protocol differs only slightly for other public key methods.

Alice now generates a 48-byte *pre-master secret*, encrypts it with Gigafirm's public key (from its certificate), and sends the result to Gigafirm, who decrypts it. Both Alice and Gigafirm now have the following secret random numbers:

1. The 32-byte random number rA that Alice sent Gigafirm.

2. The 32-byte random number rG that Gigafirm sent Alice.

3. The 48-byte pre-master secret spm.

Note that the two 32-byte numbers were not sent securely. The pre-master secret is secure, however.

Since they both have the same numbers, both Alice and Gigafirm can calculate the *master secret* as the concatenation of

MD5(spm||SHA−1(A||spm||rA||rG))MD5(spm||SHA−1(BB||spm||rA||rG))MD5(spm||SHA−1(CCC||spm||rA||rG)).

The A, BB, and CCC are strings added for padding. Note that timestamps are built into rA and rG. This prevents Eve from doing replay attacks, where she tries to use information intercepted from one session to perform similar transactions later.

Since MD5 produces a 128-bit (= 16-byte) output, the master secret has 48

bytes. The master secret is used to produce a *key block*, by the same process that the master secret was produced from the pre-master secret. Enough hashes are concatenated to produce a sufficiently long key block. The key block is then cut into six secret keys, three for communications from Alice to Gigafirm and three for communications from Gigafirm to Alice. For Alice to Gigafirm, one key serves as the secret key in the block cipher (3DES, AES, ...) chosen at the beginning of the communications. The second is a message authentication key. The third is the initial value for the CBC mode of the block cipher. The three other keys are for the corresponding purposes for Gigafirm to Alice.

Now Alice and Gigafirm are ready to communicate using the record protocol. When Alice sends a message to Gigafirm, she does the following:

1. Compresses the message using the agreed-upon compression method.

2. Hashes the compressed message together with the message authentication key (the second key obtained from the key block). This yields the hashed message authentication code.

3. Uses the block cipher in CBC mode to encrypt the compressed message together with the hashed message authentication code, and sends the result to Gigafirm.

Gigafirm now does the following:

1. Uses the block cipher to decrypt the message received. Gigafirm now has the compressed message and the hashed message authentication code.

2. Uses the compressed message and the Alice-to-

Gigafirm message authentication key to recompute the hashed message authentication code. If it agrees with the hashed message authentication code that was in the message, the message is authenticated.

3. Decompresses the compressed message to obtain Alice's message.

Communications from Gigafirm are encrypted and decrypted similarly, using the other three keys deduced from the key block. Therefore, Alice and Gigafirm can exchange information securely.

15.8 Secure Electronic Transaction

Every time someone places an order in an electronic transaction over the Internet, large quantities of information are transmitted. These data must be protected from unwanted eavesdroppers in order to ensure the customer's privacy and prevent credit fraud. Requirements for a good electronic commerce system include the following:

1. Authenticity: Participants in a transaction cannot be impersonated and signatures cannot be forged.

2. Integrity: Documents such as purchase orders and payment instructions cannot be altered.

3. Privacy: The details of a transaction should be kept secure.

4. Security: Sensitive account information such as credit card numbers must be protected.

All of these requirements should be satisfied, even over public communication channels such as the Internet.

In 1996, the credit card companies MasterCard and Visa called for the establishment of standards for electronic commerce. The result, whose development involved several companies, is called the SET, or Secure Electronic TransactionTM protocol. It starts with the existing credit card system and allows people to use it securely over open channels.

The SET protocol is fairly complex,

involving, for example, the SSL protocol in order to certify that the cardholder and merchant are legitimate and also specifying how payment requests are to be made. In the following we'll discuss one aspect of the whole protocol, namely the use of dual signatures.

There are several possible variations on the following. For example, in order to improve speed, a fast symmetric key system can be used in conjunction with the public key system. If there is a lot of information to be transmitted, a randomly chosen symmetric key plus the hash of the long message can be sent via the public key system, while the long message itself is sent via the faster

symmetric system. However, we'll restrict our attention to the simplest case where only public key methods are used.

Suppose Alice wants to buy a book entitled How to Use Other People's Credit Card Numbers to Defraud Banks, which she has seen advertised on the Internet. For obvious reasons, she feels uneasy about sending the publisher her credit card information, and she certainly does not want the bank that issued her card to know what she is buying. A similar situation applies to many transactions. The bank does not need to know what the customer is ordering, and for security reasons the merchant should not know the card

number. However, these two pieces of information need to be linked in some way. Otherwise the merchant could attach the payment information to another order. **Dual signatures** solve this problem.

The three participants in the following will be the Cardholder (namely, the purchaser), the Merchant, and the Bank (which authorizes the use of the credit card).

The Cardholder has two pieces of information:

- GSO = Goods and Service Order, which consists of the cardholder's and merchant's names, the quantities of each item ordered, the prices, etc.
- PI = Payment Instructions, including the merchant's name, the credit card number, the total

price, etc.

The system uses a public hash function; let's call it H. Also, a public key cryptosystem such as RSA is used, and the Cardholder and the Bank have their own public and private keys. Let EC, EM, and EB denote the (public) encryption functions for the Cardholder, the Merchant, and the Bank, and let DC, DM, and DB be the (private) decryption functions.

The Cardholder performs the following procedures:

1. Calculates GSOMD=H(EM(GSO)), which is the message digest, or hash, of an encryption of GSO.

2. Calculates PIMD=H(EB(PI)), which is the message digest of an encryption of PI.

3. Concatenates GSOMD and PIMD to obtain PIMD||

GSOMD, then computes the hash of the result to obtain the payment-order message digest POMD=H(PIMD||GSOMD).

4. Signs POMD by computing DS=DC(POMD). This is the dual signature.

5. Sends EM(GSO), DS, PIMD, and EB(PI) to the Merchant.

The Merchant then does the following:

1. Calculates H(EM(GSO)) (which should equal *GSOMD*).

2. Calculates H(PIMD||H(EM(GSO))) and EC(DS). If they are equal, then the Merchant has verified the Cardholder's signature and is therefore convinced that the order is from the Cardholder.

3. Computes DM(EM(GSO)) to obtain GSO.

4. Sends GSOMD, EB(PI), and DS to the Bank.

The Bank now performs the following:

1. Computes H(EB(PI)) (which should equal *PIMD*).

2. Concatenates H(EB(PI)) and GSOMD.

3. Computes HH(EB(PI))||GSOMD and EC(DS). If they are equal, the Bank has verified the Cardholder's signature.

4. Computes DB(EB(PI)), obtaining the payment instructions PI.

5. Returns an encrypted (with EM) digitally signed authorization to the Merchant, guaranteeing payment.

The Merchant completes the procedure as follows:

1. Returns an encrypted (with EC) digitally signed receipt to the Cardholder, indicating that the transaction has been completed.

The Merchant only sees the encrypted form EB(PI) of the payment instructions, and so does not see the credit card number. It would be infeasible for the Merchant or the Bank

to modify any of the information regarding the order because the hash function is used to compute DS.

The Bank only sees the message digest of the Goods and Services Order, and so has no idea what is being ordered.

The requirements of integrity, privacy, and security are met by this procedure. In actual implementations, several more steps are required in order to protect authenticity. For example, it must be guaranteed that the public keys being used actually belong to the participants as claimed, not to impostors. Certificates from a trusted authority are used for this purpose.

15.9 Exercises

1. In a network of three users, A, B, and C, we would like to use the Blom scheme to establish session keys between pairs of users. Let p=31 and let

   rA=11rB=3rC=2.

   Suppose Trent chooses the numbers

   a=8b=3c=1.

   Calculate the session keys.

2.
   1. Show that in the Blom scheme, KAB ≡ a +b(rA+rB)+crArB (mod p).

   2. Show that KAB=KBA.

   3. Another way to view the Blom scheme is by using a polynomial in two variables. Define the polynomial f(x, y)=a+b(x +y)+cxy (mod p). Express the key KAB in terms of f.

3. You (U) and I (I) are evil users on a network that

uses the Blom scheme for key establishment with k=1. We have decided to get together to figure out the other session keys on the network. In particular, suppose p=31 and rU=9, rI=2. We have received aU=18, bU=29, aI=24, bI=23 from Trent, the trusted authority. Calculate a, b, and c.

4. Here is another version of the intruder-in-the-middle attack on the Diffie-Hellman key exchange in Section 10.1. It has the "advantage" that Eve does not have to intercept and retransmit all the messages between Bob and Alice. Suppose Eve discovers that p=Mq+1, where q is an integer and M is small. Eve intercepts αx and αy as before. She sends Bob $(\alpha x)q \pmod p$ and sends Alice $(\alpha y)q \pmod p$.

    1. Show that Alice and Bob each calculate the same key K.

    2. Show that there are only M possible values for K, so Eve may find K by exhaustive search.

5. Bob, Ted, Carol, and Alice want to agree on a common key (cryptographic key, that is). They publicly choose a large prime p and a primitive root α. They privately choose random numbers b, t, c, a, respectively. Describe a protocol that

allows them to compute $K \equiv \alpha^{btca} \pmod{p}$ securely (ignore intruder-in-the-middle attacks).

6. Suppose naive Nelson tries to implement an analog of the three-pass protocol of Section 3.6 to send a key K to Heidi. He chooses a one-time pad key KN and XORs it with K. He sends $M1 = KN \oplus K$ to Heidi. She XORs what she receives with her one-time pad key KH to get $M2 = M1 \oplus KH$. Heidi sends M2 to Nelson, who computes $M3 = M2 \oplus KN$. Nelson sends M3 to Heidi, who recovers K as $M3 \oplus KH$.

    1. Show that $K = M3 \oplus KH$.

    2. Suppose Eve intercepts M1, M2, M3. How can she recover K?

Chapter 16 Digital Cash

Suppose Congressman Bill Passer is receiving large donations from his friend Phil Pockets. For obvious reasons, he would like to hide this fact, pretending instead that the money comes mostly from people such as Vera Goode. Or perhaps Phil does not want Bill to know he's the source of the money. If Phil pays by check, well-placed sources in the bank can expose him. Similarly, Congressman Passer cannot receive payments via credit card. The only anonymous payment scheme seems to be cash.

But now suppose Passer has remained in

office for many terms and we are nearing the end of the twenty-first century. All commerce is carried out electronically. Is it possible to have electronic cash? Several problems arise. For example, near the beginning of the twenty-first century, photocopying money was possible, though a careful recipient could discern differences between the copy and the original. Copies of electronic information, however, are indistinguishable from the original. Therefore, someone who has a valid electronic coin could make several copies. Some method is needed to prevent such double spending. One idea would be for a central bank to have records of every coin and who has each one. But if coins are recorded as they

are spent, anonymity is compromised. Occasionally, communications with a central bank could fail temporarily, so it is also desirable for the person receiving the coin to be able to verify the coin as legitimate without contacting the bank during each transaction.

T. Okamoto and K. Ohta [Okamoto-Ohta] list six properties a digital cash system should have:

1. The cash can be sent securely through computer networks.

2. The cash cannot be copied and reused.

3. The spender of the cash can remain anonymous. If the coin is spent legitimately, neither the recipient nor the bank can identify the spender.

4. The transaction can be done *off-line*, meaning no communication with the central bank is needed during the transaction.

5. The cash can be transferred to others.

6. A piece of cash can be divided into smaller amounts.

Okamoto and Ohta gave a system that satisfies all these requirements. Several systems satisfying some of them have been devised by David Chaum and others. In Section 16.2, we describe a system due to S. Brands [Brands] that satisfies 1 through 4. We include it to show the complicated manipulations that are used to achieve these goals. But an underlying basic problem is that it and the Okamoto-Ohta system require a central bank to set up the system. This limits their use.

In Section 16.3, we give an introduction to Bitcoin, a well-known digital

currency. By ingeniously relaxing the requirements of Okamoto-Ohta, it discarded the need for a central bank and therefore became much more widely used than its predecessors.

16.1 Setting the Stage for Digital Economies

People have always had a need to engage in trade in order to acquire items that they do not have. The history of commerce has evolved from the ancient model of barter and exchange to notions of promises and early forms of credit ("If you give me bread today, then I'll give you milk in the future"), to the use of currencies and cash, to the merging of currencies and credit in the form of credit cards. All of these have changed over time, and recent shifts toward conducting transactions electronically have forced these technologies to evolve significantly.

Perhaps the most dominant way in which electronic transactions take place is with the use of credit cards. We are all familiar with making purchases at stores using conventional credit cards: our items are scanned at a register, a total bill is presented, and we pay by swiping our card (or inserting a card with a chip) at a credit card scanner. Using credit cards online is not too different. When you want to buy something online from eVendor, you enter your credit card number along with some additional information (e.g., the CVC, expiration date, address). Whether you use a computer, a smartphone, or any other type of device, there is a secure communication protocol working behind the scenes that sends this

information to eVendor, and secure protocols support eVendor by contacting the proper banks and credit card agencies to authorize and complete your transaction.

While using credit cards is extremely easy, there are problems with their use in a world that has been increasingly digital. The early 21st century has seen many examples of companies being hacked and credit card information being stolen. A further problem with the use of credit cards is that companies have the ability to track customer purchases and preferences and, as a result, issues of consumer privacy are becoming more prevalent.

There are alternatives to the use of

credit cards. One example is the introduction of an additional layer between the consumer and the vendor. Companies such as PayPal provide such a service. They interact with eVendor, providing a guarantee that eVendor will get paid, while also ensuring that eVendor does not learn who you are. Of course, such a solution begs the question of how much trust one should place in these intermediate companies.

A different alternative comes from looking at society's use of hard, tangible currencies. Coins and cash have some very nice properties when one considers their use from a security and privacy perspective. First, since they are physical objects representing real value,

they provide an immediate protection against any defaulting or credit risk. If Alice wants an object that costs five dollars and she has five dollars, then there is no need for a credit card or an I-Owe-You. The vendor can complete the transaction knowing that he or she has definitely acquired five dollars. Second, cash and coins are not tied to the individual using them, so they provide strong anonymity protection. The vendor doesn't care about Alice or her identity; it only cares about getting the money associated with the transaction. Likewise, there are no banks directly involved in a transaction. Currency is printed by a central government, and this currency is somehow backed by the government in one way or another.

Cash and coins also have the nice property that they are actually exchanged in a transaction. By this, we mean that when Alice spends her five dollars, she has handed over the money and she is no longer in possession of this money. This means that she can't spend the same money over and over. Lastly, because of the physical nature of cash and coins, there is no need to communicate with servers and banks to complete a transaction, which allows for transactions to be completed off-line.

In this chapter, we will also discuss the design of digital currencies, starting from one of the early models for digital coins and then exploring the more recent forms of cryptocurrencies by examining

the basic cryptographic constructions used in Bitcoin. As we shall see, many of the properties that we take for granted with cash and coins have been particularly difficult to achieve in the digital world.

16.2 A Digital Cash System

This section is not needed for the remaining sections of the chapter. It is included because it has some interesting ideas and it shows how hard it is to achieve the desired requirements of a digital currency.

We describe a system due to S. Brands [Brands]. The reader will surely notice that it is much more complicated than the centuries-old system of actual coins. This is because, as we mentioned previously, electronic objects can be reproduced at essentially no cost, in contrast to physical cash, which has usually been rather difficult to

counterfeit. Therefore, steps are needed to catch electronic cash counterfeiters. But this means that something like a user's signature needs to be attached to an electronic coin. How, then, can anonymity be preserved? The solution uses "restricted blind signatures." This process contributes much of the complexity to the scheme.

16.2.1 Participants

Participants are the Bank, the Spender, and the Merchant.

16.2.2 Initialization

Initialization is done once and for all by some central authority. Choose a large prime p such that $q=(p-1)/2$ is also

prime (see Exercise 15 in Chapter 13). Let g be the square of a primitive root mod p. This implies that $g^{k1} \equiv g^{k2} \pmod{p} \Leftrightarrow k1 \equiv k2 \pmod{q}$. Two secret random exponents are chosen, and g1 and g2 are defined to be g raised to these exponents mod p. These exponents are then discarded (storing them serves no useful purpose, and if a hacker discovers them, then the system is compromised). The numbers

$g, g1, g2$

are made public. Also, two public hash functions are chosen. The first, H, takes a 5-tuple of integers as input and outputs an integer mod q. The second, H0, takes a 4-tuple of integers as input and outputs an integer mod q.

16.2.3 The Bank

The bank chooses its secret identity number x and computes

$$h \equiv g^x \pmod{p}.$$

The number h is made public and identifies the bank.

16.2.4 The Spender

The Spender chooses a secret identity number u and computes the account number

$$I \equiv g^{1u} \pmod{p}.$$

The number I is sent to the Bank, which stores I along with information identifying the Spender (e.g., name, address). However, the Spender does not send u to the bank. The Bank sends

$$z' \equiv (Ig2)x \pmod{p}$$

to the Spender.

16.2.5 The Merchant

The Merchant chooses an identification number M and registers it with the bank.

16.2.6 Creating a Coin

The Spender contacts the bank, asking for a coin. The bank requires proof of identity, just as when someone is withdrawing classical cash from an account. All coins in the present scheme have the same value. A coin will be represented by a 6-tuple of numbers

$$(A, B, z, a, b, r).$$

This may seem overly complicated, but

we'll see that most of this effort is needed to preserve anonymity and at the same time prevent double spending.

Here is how the numbers are constructed.

1. The Bank chooses a random number w (a different number for each coin), computes

   $gw \equiv gw$ and $\beta \equiv (Ig2)w \pmod p$,

   and sends gw and β to the Spender.

2. The Spender chooses a secret random 5-tuple of integers

   $(s, x1, x2, \alpha1, \alpha2)$.

3. The Spender computes

   $A \equiv (Ig2)s$, $B \equiv g1x1g2x2$, $z \equiv z$
   $'s$, $a \equiv gw\alpha1g\alpha2$, $b \equiv \beta s\alpha1A\alpha2 \pmod p$.

   Coins with A=1 are not allowed. This can happen in only two ways. One is when $s \equiv 0 \pmod q$, so we require $s \equiv 0$. The other is when $Ig2 \equiv 1 \pmod p$, which means the Spender has solved a discrete logarithm problem by a lucky

choice of u. The prime p should be chosen so large that this has essentially no chance of happening.

4. The Spender computes

   $c \equiv \alpha 1 - 1 H(A, B, z, a, b) (\text{mod } q)$

   and sends c to the Bank. Here H is the public hash function mentioned earlier.

5. The Bank computes $c1 \equiv cx+w \text{ (mod } q)$ and sends c1 to the Spender.

6. The Spender computes

   $r \equiv \alpha 1 c1 + \alpha 2 (\text{mod } q)$.

   The coin (A, B, z, a, b, r) is now complete. The amount of the coin is deducted from the Spender's bank account.

The procedure, which is quite fast, is repeated each time a Spender wants a coin. A new random number w should be chosen by the Bank for each transaction. Similarly, each spender should choose a new 5-tuple

(s, x1, x2, α1, α2) for each coin.

16.2.7 Spending the Coin

The Spender gives the coin
(A, B, z, a, b, r) to the Merchant. The following procedure is then performed:

1. The Merchant checks whether

   $$gr \equiv a^{hH(A, B, z, a, b)} A^r \equiv z^{H(A, B, z, a, b)b} \pmod{p}.$$

   If this is the case, the Merchant knows that the coin is valid. However, more steps are required to prevent double spending.

2. The Merchant computes

   $$d = H0(A, B, M, t),$$

   where H0 is the hash function chosen in the initialization phase and t is a number representing the date and time of the transaction. The number t is included so that different transactions will have different values of d. The Merchant sends d to the Spender.

3. The Spender computes

$r1 \equiv dus+x1,\ r2 \equiv ds+x2 \pmod{q}$,

where u is the Spender's secret number, and s, x1, x2 are part of the secret random 5-tuple chosen earlier. The Spender sends r1 and r2 to the Merchant.

4. The Merchant checks whether

   $g1r1g2r2 \equiv AdB \pmod{p}$.

   If this congruence holds, the Merchant accepts the coin. Otherwise, the Merchant rejects it.

16.2.8 The Merchant Deposits the Coin in the Bank

A few days after receiving the coin, the Merchant wants to deposit it in the Bank. The Merchant submits the coin (A, B, z, a, b, r) plus the triple (r1, r2, d). The Bank performs the following:

1. The Bank checks that the coin (A, B, z, a, b, r) has

not been previously deposited. If it hasn't been, then the next step is performed. If it has been previously deposited, the Bank skips to the Fraud Control procedures discussed in the next subsection.

2. The Bank checks that

   $$gr \equiv ahH(A, B, z, a, b)Ar \equiv zH(A, B, z, a, b)b, \text{ and } g1r1g2r2 \equiv AdB$$

   If so, the coin is valid and the Merchant's account is credited.

16.2.9 Fraud Control

There are several possible ways for someone to try to cheat. Here is how they are dealt with.

1. The Spender spends the coin twice, once with the Merchant, and once with someone we'll call the Vendor. The Merchant submits the coin along with the triple (r1, r2, d). The Vendor submits the coin along with the triple (r1′, r2′, d′). An easy calculation shows that

   $$r1 - r1' \equiv us(d - d'), \; r2 - r2' \equiv s(d - d') \pmod{q}.$$

Dividing yields $u \equiv (r1 - r1')(r2 - r2')^{-1} \pmod{q}$. The Bank computes $I \equiv g1^u \pmod{p}$ and identifies the Spender. Since the Bank cannot discover u otherwise, it has proof (at least beyond a reasonable doubt) that double spending has occurred. The Spender is then sent to jail (if the jury believes that the discrete logarithm problem is hard).

2. The Merchant tries submitting the coin twice, once with the legitimate triple (r1, r2, d) and once with a forged triple (r1′, r2′, d′). This is essentially impossible for the Merchant to do, since it is very difficult for the Merchant to produce numbers such that

   $g1^{r1'} g2^{r2'} \equiv A d' B \pmod{p}$.

3. Someone tries to make an unauthorized coin. This requires finding numbers such that $gr \equiv a^{hH(A, B, z, a, b)}$ and $Ar \equiv z^{H(A, B, z, a, b)} b$. This is probably hard to do. For example, starting with A, B, z, a, b, then trying to find r, requires solving a discrete logarithm problem just to make the first equation work. Note that the Spender is foiled in attempts to produce a second coin using a new 5-tuple since the values of x is known only to the Bank. Therefore, finding the correct value of r is very difficult.

4. Eve L. Dewar, an evil merchant, receives a coin from the Spender and deposits it in the bank, but also tries to spend the coin with the Merchant. Eve gives the coin to the Merchant, who computes d′, which very likely is not equal to d. Eve does not know u, x1, x2, s, but she must choose r1′ and r2′ such that $g1r1'g2r2' \equiv Ad'B \pmod p$. This again is a type of discrete logarithm problem. Why can't Eve simply use the r1, r2 that she already knows? Since $d' \neq d$, the Merchant would find that $g1r1g2r2 \equiv Ad'B$.

5. Someone working in the Bank tries to forge a coin. This person has essentially the same information as Eve, plus the identification number I. It is possible to make a coin that satisfies $gr \equiv ahH(A, B, z, a, b)$. However, since the Spender has kept u secret, the person in the bank will not be able to produce a suitable r1. Of course, if s=0 were allowed, this would be possible; this is one reason A=1 is not allowed.

6. Someone steals the coin from the Spender and tries to spend it. The first verification equation is still satisfied, but the thief does not know u and therefore will not be able to produce r1, r2 such that $g1r1g2r2 \equiv Ad'B$.

7. Eve L. Dewar, the evil merchant, steals the coin and (r1, r2, d) from the Merchant before they are

submitted to the Bank. Unless the bank requires merchants to keep records of the time and date of each transaction, and therefore be able to reproduce the inputs that produced d, Eve's theft will be successful. This, of course, is a flaw of ordinary cash, too.

16.2.10 Anonymity

During the entire transaction with the Merchant, the Spender never needs to provide any identification. This is the same as for purchases made with conventional cash. Also, note that the Bank never sees the values of A, B, z, a, b, r for the coin until it is deposited by the Merchant. In fact, the Bank provides only the number w and the number c1, and has seen only c. However, the coin still contains information that identifies the Spender in the case of double spending. Is it

possible for the Merchant or the Bank to extract the Spender's identity from knowledge of the coin (A, B, z, a, b, r) and the triple (r1, r2, d)? Since the Bank also knows the identification number I, it suffices to consider the case where the Bank is trying to identify the Spender. Since s, x1, x2 are secret random numbers known only to the Spender, A and B are random numbers. In particular, A is a random power of g and cannot be used to deduce I. The number z is simply Ax (mod p), and so does not provide any help beyond what is known from A. Since a and b introduce two new secret random exponents $\alpha 1$, $\alpha 2$, they are again random numbers from the viewpoint of everyone except the Spender.

At this point, there are five numbers, A, B, z, a, b, that look like random powers of g to everyone except the Spender. However, when $c \equiv \alpha 1 - 1H(A, B, z, a, b) (\mathrm{mod}\ q)$ is sent to the Bank, the Bank might try to compute the value of H and thus deduce $\alpha 1$. But the Bank has not seen the coin and so cannot compute H. The Bank could try to keep a list of all values c it has received, along with values of H for every coin that is deposited, and then try all combinations to find $\alpha 1$. But it is easily seen that, in a system with millions of coins, the number of possible values of $\alpha 1$ is too large for this to be practical. Therefore, it is unlikely that knowledge of c, hence of b, will help the Bank identify the Spender.

The numbers $\alpha 1$ and $\alpha 2$ provide what Brands calls a **restricted blind signature** for the coin. Namely, using the coin once does not allow identification of the signer (namely, the Spender), but using it twice does (and the Spender is sent to jail, as pointed out previously).

To see the effect of the restricted blind signature, suppose $\alpha 1$ is essentially removed from the process by taking $\alpha 1=1$. Then the Bank could keep a list of values of c, along with the person corresponding to each c. When a coin is deposited, the value of H would then be computed and compared with the list. Probably there would be only one person for a given c, so the Bank could

identify the Spender.

16.3 Bitcoin Overview

In this section we provide a brief overview of Bitcoin. For those interested in the broader issues behind the design of cryptocurrencies like Bitcoin, we refer to the next section.

Bitcoin is an example of a ledger-based cryptocurrency that uses a combination of cryptography and decentralized consensus to keep track of all of the transactions related to the creation and exchange of virtual coins, known as bitcoins.

Bitcoin is a very sophisticated collection of cryptography and communication

protocols, but the basic structure behind the operation of Bitcoin can be summarized as having five main stages:

- Users maintain a transaction ledger;
- Users make transactions and announce their transactions;
- Users gather transactions into blocks;
- Users solve cryptographic puzzles using these blocks;
- Users distribute their solved puzzle block.

Let’s start in the middle. There are many users. Transactions (for example, Alice gives three coins to Bob and five coins to Carla) are happening everywhere. Each transaction is broadcast to the network. Each user collects these transactions and verifies that they are legitimate. Each user collects the valid

transactions into a block. Suddenly, one user, say Zeno, gets lucky (see “Mining” below). That user broadcasts this news to the network and gets to add his block to the ledger that records all transactions that have ever taken place. The transactions continue throughout the world and continue to be broadcast to everyone. The users add the valid transactions to their blocks, possibly including earlier transactions that were not included in the block that just got added to the ledger. After approximately 10 minutes, another user, let’s say Xenia, gets lucky and is allowed to add her block to the ledger. If Xenia believes that all of the transactions are valid in the block that Zeno added, then Xenia adds her block to the ledger that

includes Zeno's block. If not, then Xenia adds her block to the ledger that was in place before Zeno's block was added. In either case, Xenia broadcasts what she did.

Eventually, after approximately another 10 minutes, Wesley gets lucky and gets to add his block to the ledger. But what if there are two or more competing branches of the ledger? If Wesley believes that one contains invalid transactions, he does not add to it, and instead chooses among the remaining branches. But if everything is valid, then Wesley chooses the longest branch. In this way, the network builds a consensus as to the validity of transactions. The longer branch has had more randomly

chosen users certify the transactions it contains.

Stopping Double Spending. Now, suppose Eve buys something from the vendor Venus and uses the same coins to buy something from the seller Selena. Eve broadcasts two transactions, one saying that she paid the coins to Venus and one saying that she paid the coins to Selena. Some users might add one of the transactions to their blocks, and some add the other transaction to their blocks. There is possibly no way someone can tell which is legitimate. But eventually, say, Venus's block ends up in a branch of blocks that is longer than the branch containing Selena's block. Since this branch is longer, it keeps being

augmented by new blocks, and the payment to Selena becomes worthless (the other transactions in the block could be included in later additions to the longer branch). What happens to Selena? Has she been cheated? No. After concluding the deal with Eve, Selena waits an hour before delivering the product. By that time, either her payment has been included in the longer branch, or she realizes that Eve's payment to her is worthless, so Selena does not deliver the product.

Incentives. Whenever a user gets lucky and is chosen to add a block to the ledger, that user collects fees from each transaction that is included in the block. These payments of fees are listed as

transactions that form part of the block that is added to the ledger. If the user includes invalid transactions, then it is likely that a new branch will soon be started that does not include this block, and thereby the payments of transaction fees become worthless. So there is an incentive to include many transactions, but there is also an incentive to verify their validity. At present, there is also a reward for being the lucky user, and this is included as a payment in the user's block that is being added to the ledger. After every 210000 blocks are added to the ledger, which takes around four years at 10 minutes per block, the reward amount is halved. In 2018, the reward stood at 25 bitcoins. The overall system is set up so that there will

eventually be a total of 21 million bitcoins in the system. After that, the plan is to maintain these 21 million coins as the only bitcoins, with no more being produced. At that point, the transaction fees are expected to provide enough incentive to keep the system running.

Mining. How is the lucky user chosen? Each user computes

h(Nonce||prevhash||TransX1||TransX2|···||TransXN)

for billions of values of *Nonce*. Here, h is the hash function SHA-256, *Nonce* is a random bitstring to be found, *prevhash* is the hash of the previous block in the blockchain, TransXj are the transactions that the user is proposing to add to the ledger. On the average, after around

1020 hashes are computed worldwide, some user obtains a hash value whose first 66 binary digits are 0s. (These numbers are adjusted from time to time as more users join in order to keep the average spacing at 10 minutes.) This user is the “lucky” one. The nonce that produced the desired hash is broadcast, along with the hash value obtained and the block that is being added to the ledger. The mining then resumes with the updated ledger, at least by users who deem the new block to be valid.

Mining uses enormous amounts of electricity and can be done profitably only when inexpensive electric power is available. When this happens, massive banks of computers are used to compute

hash values. The rate of success is directly proportional to the percentage of computer power one user has in relation to the total power of all the users in the world. As long as one party does not have access to a large fraction of the total computing power, the choice of the lucky user will tend to be random enough to prevent cheating by a powerful user.

16.3.1 Some More Details

Users Maintain a Transaction Ledger: The basic structure behind Bitcoin is similar to many of the other ledger-based cryptocurrencies. No actual *digital coins* are actually exchanged. Rather, a ledger is used to keep track of transactions that take place, and pieces

of the ledger are the digital objects that are shared. Each user maintains their own copy of the ledger, which they use to record the community's collection of transactions. The ledger consists of blocks, structured as a blockchain (see Section 12.7), which are cryptographically signed and which reference previous blocks in the ledger using hash pointers to the previous block in the ledger.

Also, a transaction includes another hash pointer, one to the transaction that says that the spender has the bitcoins that are being spent. This means that when someone else checks the validity of a transaction, it is necessary to look at only the transactions that occurred since

that earlier transaction. For example, if George posts a transaction on June 14 where he gives 13 bitcoins to Betsy, George includes a pointer to the transaction on the previous July 4 where Tom paid 18 bitcoins to George. When Alex wants to check the validity of this transaction, he checks only those transactions from July 4 until June 14 to be sure that George didn't also give the bitcoins to Aaron. Moreover, George also can post a transaction on June 14 that gives the other five bitcoins from July 4 to himself. In that way, the ledger is updated in a way that there aren't small pieces of long-ago transactions lying around.

Making and Announcing Transactions:

A transaction specifies the coins from a previous transaction that are being consumed as input. As output of the transaction, it specifies the address of the recipients and the amount of coins to be delivered to these recipients. For example, in addresses for Bitcoin Version 1, each user has a public/private pair of keys for the Elliptic Curve Digital Signature Algorithm. The user's address is a 160-bit cryptographic hash of their public key. The 160-bit hash is determined by first calculating the SHA-256 hash of the user's public key, and then calculating the RIPEMD-160 hash (this is another widely used hash function) of the SHA-256 output. A four-byte cryptographic checksum is added to the 160-bit hash, which is

calculated from SHA-256 being applied twice to the 160-bit hash. Finally, this is encoded into an alphanumeric representation.

The transaction is signed by the originator of the transaction using their private key, and finally it is announced to the entire set of users so it can be added to blocks that will be appended to the community's ledger.

Gathering Transactions into Blocks: Transactions are received by users. Users verify the transactions that they receive and discard any that they are not able to verify. There are several reasons why transactions might not verify. For example, since the communications are taking place on a network, it is possible

that not every user will receive the same set of transactions at the same time. Or, users might be malicious and attempt to announce false transactions, and thus Bitcoin relies on users to examine the collection of new transactions and previous transactions to ensure that no malicious behavior is taking place (such as double spending, or attempting to steal coins). Users then gather the transactions they believe are valid into the block that they are forming. A new, candidate block consists of a collection of transactions that a user believes is valid, and these transactions are arranged in a Merkle Tree to allow for efficient searching of transactions within a block. The block also contains a hash pointer to a previous block in the ledger.

The hash pointer is calculated using the SHA-256 hash of a previous block.

Anonymity: Any cash system should have some form of anonymity. In Bitcoin, a user is identified only through that user's public key, not through the actual identity. A single user could register under multiple names so that an observer will not see several transactions being made by one user and deduce the user's identity. Of course, some anonymity is lost because this user will probably need to make transfers from the account of one of his names to the account of another of his names, so a long-term analysis might reveal information.

An interesting feature of Bitcoin is that

registering under multiple names does not give a user more power in the mining operations because the computational resources for mining will be spread over several accounts but will not increase in power. Therefore, the full set of this user's names has the same probability of being lucky as if the user had opened only a single account. This is much better than the alternative where a consensus could be reached by voting with one vote per account.

16.4 Cryptocurrencies

In this section, we give a general discussion of issues related to digital cash, using Bitcoin as an example.

The Brands digital cash system from Section 16.2 gives insight into how difficult it can be to re-create the benefits of cash using just cryptography and, even with all its cryptographic sophistry, the Brands scheme is still not able to allow for the cash to be transferred to others or for the digital money to be divided into smaller amounts. Up until the 2000s, almost all digital cash protocols faced many shortcomings when it came to creating

digital objects that acted like real-world money. For example, in order to allow for anonymity and to prevent double spending, it was necessary to introduce significant complexity into the scheme. Some protocols were able to work only online (i.e., they required the ability to connect to an agent acting as a bank to verify the validity of the digital currency), some protocols like Brands's system did not allow users to break cash into smaller denominations and thereby issue change in transactions. Beyond the technical challenges, some systems, like Chaum's ECash, faced pragmatic hurdles in persuading banks and merchants to adopt their system. Of course, without merchants to buy things from, there won't be users to make

purchases, and this can be the death of a cryptocurrency.

One of the major, practical challenges that the early forms of cryptocurrencies faced was the valuation of their digital cash. For example, just because a digital coin is supposedly worth $100, what actually makes it worth that amount? Many early digital currencies attempted to tie themselves to real-world currencies or to real-world commodities (like gold). While this might seem like a good idea, it also introduced several practical challenges: it implicitly meant that the value of the entire cryptocurrency had to be backed by an equivalent amount of real-world cash or coins. And this was a problem – if these

new, digital currencies wanted to exist and be used, then they had to acquire a lot of real-world money to back them.

This also meant that the new forms of digital cash weren't actually their own currency, but actually just another way to spend an existing currency or commodity. Since needing real-world assets in order to get a cryptocurrency off the ground was a hurdle, it led many to ask "What if one could make the digital cash its own currency that was somehow valued independently from other currencies?"

It turns out that many of the technical solutions needed to overcome the various hurdles that we have outlined already existed in the early 2000s and

what was needed was a different perspective. In 2008, Satoshi Nakamoto presented a landmark paper [Nakamoto], which launched the next generation of cryptocurrencies and introduced the well known cryptocurrency Bitcoin. These cryptocurrencies, which include Bitcoin, Ethereum, and many others, have been able to overcome many of the hurdles that stifled previous attempts such as ECash. Currently, currencies like Bitcoin and Ethereum are valid currencies that are traded internationally.

The new generation of cryptocurrencies were successful because they made practical compromises or, to put it

another way, they did not try to force all of the properties of real-world cash onto digital currencies. For example, a digital currency like Bitcoin does not work offline, but it does cleverly use decentralization to provide robustness so that if parts of the system are offline, the rest of the system can continue to function. Another major paradigm shift was the realization that one did not need to create a digital object corresponding to a coin or to cash, but rather that the money could be *virtual* and one only needed to keep track of the trading and exchange of this virtual money. That is, bookkeeping was what was important, not the coin itself! And, once we stop trying to create a digital object that acts like cash, then it becomes a lot easier to

achieve properties like preventing double spending and being able to divide coins into smaller parts. Another paradigm shift was the realization that, rather than tying the value of the currency to real-world currency, one could tie the value to solving a hard (but not impossible) computational problem and make the solution of that computational problem have value in the new currency.

In order to understand how these new cryptocurrencies work, let's take a look at the basic structures being used in Bitcoin. Many of the other currencies use similar building blocks for their design.

As we just mentioned, one of the major

changes between Bitcoin and older digital cash schemes is a move away from a digital object representing a coin and instead to the use of an imaginary or virtual currency that is kept track of in bookkeeping records. These ledgers, as they are known, record the creation and exchange of these virtual coins. The use of cryptography to form secure ledgers has become a growing trend that has impacted many applications beyond the support of cryptocurrencies.

Let us look at a simple ledger. To start, assume that we have a central bank, which we call BigBank. Later we shall remove BigBank, but for now it is useful to have BigBank around in order to start the discussion. We suppose that

BigBank can make new virtual coins, and that it wants to share them with Alice. To do so, it makes a ledger that records these events, something that looks like:

Ledger_BigBank: Create 100 coinsBigBank → Alice: 100 coins

This ledger can be shared with Alice or anyone else, and after looking at it, one can easily figure out the sequence of events that happened. BigBank made 100 coins, and then these coins were given by BigBank to Alice. Assuming there have not been any other transactions, then one can conclude that Alice is now the owner of 100 virtual coins.

Of course, Alice should not just trust this ledger. Anyone could have made

this ledger, pretending to be BigBank. As it is, there is nothing tying this ledger to its creator. To solve this problem, we must use digital signatures. Therefore, BigBank has a public–private key pair and has shared its public key with Alice and the rest of the world. There are many ways in which this could have been done, but it is simplest to think of some certificate authority issuing a certificate containing BigBank's public key credentials. Now, rather than share only the Ledger, BigBank also shares signBB(Ledger).

This makes it harder for someone to pretend to be BigBank, but we still have some of the usual problems that we encountered in Chapter 15. For

example, one problem that stands out is that it is possible to perform a replay attack. If Alice receives five separate copies of the signed Ledger, then she might conclude that she has 500 coins. Therefore, in order to fix this we need to use some form of unique transaction identifier or counter in Ledger before BigBank signs it, something like:

Ledger 1_TID-1.BigBank: Create 100 coinsTID-2.BigBank → Alice: 100 coins

Now, the signed Ledger allows anyone to have some trust that BigBank has given Alice 100 coins.

Alice's coins aren't much use to her unless she can spend them. Suppose she goes to Sarah's Store, which is known to accept BigBank's coins, and wants to

buy an item using BigBank's coins. Alice could write another entry in the ledger by giving Sarah's Store an update to the ledger that looks like

Ledger 2_TID-3.Alice → SarahStore: 100 coins

Now, Alice signs this update and sends Sarah's Store signAlice(Ledger 2). Sarah's Store can indeed verify that Alice made this update, but a problem arises: How does Sarah's Store know that Alice in fact had 100 coins to spend?

A simple way to take care of this is for Alice to attach a signed copy of BigBank's original Ledger. Now Sarah's Store can verify that Alice did get 100 coins from BigBank.

This works, but we come to classic double spending problem that all digital currencies must address. Right now, there is nothing preventing Alice from going to Vendor Veronica and spending those 100 coins again. Alice can simply make another version of Ledger 2, call it Ledger 2′:

Ledger 2′_TID-3.Alice → Vendor Veronica: 100 coins

She can sign Ledger 2′ and send both it and BigBank's signed copy of the original Ledger 1. But even with these, there is no way for Vendor Veronica to be assured that Alice did not spend her 100 coins elsewhere, which she in fact did.

What is needed is a way to solve the double spending problem. We can do

this by slightly revising the purchasing protocol to place BigBank in a more central role. Rather than let Alice announce her transaction, we require Alice to contact BigBank when she makes purchases and BigBank to announce updates to the ledger. The ledger of transactions operates as an append-only ledger in that the original ledger with all of its updates contains the history of all transactions that have ever occurred using BigBank's coins. Since it is not possible to remove transactions from the ledger, it is possible for everyone to see if double spending occurs simply by keeping track of the ledger announced by BigBank.

Ensuring that this append-only ledger is cryptographically protected requires that each update is somehow tied to previous versions of the ledger updates, otherwise it might be possible for transactions to be altered. The construction of the append-only ledger is built using a cryptographic data structure known as a blockchain, which we introduced in Section 12.7. A blockchain consists of two different forms of hash-based data structures. First, it uses hash pointers in a hash chain, and second it uses a Merkle tree to store transactions. The use of the hash chain in the blockchain makes the ledger tamper-proof. If an adversary attempts to tamper with data that is in block k, then the hash contained in block k+1, namely, the

hash of the correct block k, will not match the hash of the altered block k. The use of the Merkle tree provides an efficient means of determining whether a transaction belongs to a block.

Now suppose that BigBank wants to keep track of all of the transactions that have taken place and publish it so that others can know these transactions. BigBank could periodically publish blocks, or it could gather an entire collection of blocks and publish them all at once. Ultimately, BigBank publishes all of the blocks of the blockchain along with a final hash pointer that certifies the entire collection of blocks in the blockchain. Each of the blocks in the blockchain, along with the final hash

pointer, is signed by BigBank so that others know the identity of the entity publishing the ledger. Thus, the hash chaining in the blockchain provides data integrity, while the digital signatures applied to each block provide origin authentication.

Let us return to the story of BigBank and all of the various participants who might want to use or acquire some of BigBank's digital coins. Everyone wishing to use some of their BigBank coins must tell BigBank how many coins or how much value is being transferred for a purchase and whom the coins will be given to. BigBank does not need to know what is being purchased, or why there is a transaction occurring;

it just needs to know the value of the exchange and the recipient. BigBank will then gather several of these individual transactions into a single block, and publish that block as an update to the blockchain that represents BigBank's ledger.

To do this, though, BigBank needs a way to describe the transactions in the ledger, and this means that there needs to be a set of basic transaction types that are allowed. It turns out that we don't need very many to have a useful system. For example, it is useful for an entity like BigBank to create coins, where each coin has a value, a serial number, and a recipient. BigBank can, in fact, create as many coins as it wants during

the time before it publishes the next block in the blockchain. For example, suppose BigBank creates three coins with different recipients, and that it publishes the creation of these coins in the data portion of a block that looks like:

BlockID: 76 CreateCoins		
TransID	Value	Recipient
0	5	PK_{BB}
1	2	PK_{Alice}
2	10	PK_{Bob}

16.4-1 Full Alternative Text

When this block is published, it informs the rest of the world that BigBank created several coins of different values. First, in transaction 0 for block 76, a coin worth 5 units was created and given to PKBB (which, as will be

explained shortly, stands for BigBank's Public Key). This coin will be referred to as coin 76(0) since it was made in block 76, transaction 0. Similarly, a coin was created in transaction 1 of Block 76 for PKAlice, which will be referred to as coin 76(1). Lastly, a coin was created in transaction 2 for PKBob, and will be referred to as coin 76(2). Who or what are PKBB, PKAlice, and PKBob? This is an example of something clever that Bitcoin borrowed from other applications of cryptography. In [HIP], it was recognized that the notion of one's identity is best tied to something that only that entity should possess. This is precisely what a private key provides in public key cryptography, and so if someone wants to send something to

Alice or Bob, then they can use Alice or Bob's public key (which they know because the public key is public) as the name of the recipient.

In order to support purchases made by Alice and others, we need to allow for the users of BigBank currency to consume coins in exchange for goods, and to receive change for their transactions. It was realized that it was easier to simply destroy the original coin and issue new coins as payment to the vendors and new coins as change to those making purchases than it was to break a coin down into smaller denominations, which had proven difficult to accomplish for previous digital coin schemes. So, if Bob has a

coin 76(2) worth 10 units, and he wants to buy something worth 7 units, BigBank destroys 76(2), issues a new coin to the vendor worth 7 units, and issues a new coin to Bob worth 3 units.

Of course, BigBank is big and might need to handle many transactions during the time it takes to form a block. Therefore, BigBank needs to hear from all of its customers about their purchases, and it needs to make certain that the coins that its customers are spending are valid and not previously spent. It also needs to create new coins and give the new coins to those who are selling to BigBank's customers, while also issuing new coins as change. Let us look at how this could work. Suppose

Alice has a coin 41(2) that is worth 20 units, and that she wants to buy a widget from Sarah's Store for 15 units. Then, the first step that BigBank takes is to destroy Alice's coin worth 20 units, then issue a new coin to Sarah's Store worth 15 units and a new coin to Alice worth 5 units, which corresponds to Alice's change from her purchase. Thus, one coin was consumed and two were created, but the total value of the coins being consumed is equal to the total value of the coins being created from the transaction. Lastly, before BigBank can publish its update to the ledger, it needs to get all of the owners of the coins that were consumed to sign the transaction, thereby ensuring that they knowingly spent their coins. Any good

cryptographic signature scheme would work. Bitcoin uses the Elliptic Curve Digital Signature Algorithm (ECDSA), which we will see in Exercise 24 in Chapter 21.

The next block of transactions in the blockchain thus looks like:

BlockID: 77 ConsumeCoins: 41(2), 16(0), 31(1) CreateCoins		
TransID	Value	Recipient
0	15	$PK_{SarahStore}$
1	5	PK_{Alice}
2	4	PK_{Bob}
3	11	$PK_{Charles}$
Signatures		

16.4-2 Full Alternative Text

BigBank's approach to managing its ledger of transactions seems to work well, but it has one major problem, and that is the fact that BigBank has to be

involved in every transaction. Not only does this mean a lot of work for BigBank, but it also leads to several security concerns, such as the risk of a denial of service to the system if BigBank becomes disconnected from the rest of the network, or the risk that BigBank might use its influence as the central entity in the economy to extract extortion from its members. Although BigBank cannot create false transactions, it could ask for large *service* fees in order to carry out a transaction.

Bitcoin gets around the problem of a central bank vetting each and every transaction by making the blockchain protocol decentralized. In other words,

rather than having a single entity check the transactions, Bitcoin uses the notion of community consensus to check that the transactions are correct. Also, rather than have a single entity maintain and update the ledger of transactions, Bitcoin allows the members of the community to form the next block in the blockchain and thereby update the ledger themselves. In fact, Bitcoin went further in that it did not trust banks to mint coins themselves. Instead, Bitcoin devised a means by which coins get created periodically and are awarded to those who do work to keep the Bitcoin system moving smoothly.

In order to understand this, we need to set the stage for how life in the Bitcoin

universe works. Every T units of time (in Bitcoin, T=10 minutes), the users in the Bitcoin network must agree on which transactions were broadcast by the users making purchases and trading in bitcoins. Also, the nodes must agree on the order of the transactions, which is particularly important when money is being exchanged for payments. Bitcoin then proceeds in rounds, with a new update to the ledger being published each round.

Every user in the Bitcoin network keeps track of a ledger that contains all of the blocks that they've agreed upon. Every user also keeps a list of transactions that they have heard about but have not yet been added to the blockchain. In

particular each user might have slightly different versions of this list, and this might happen for a variety of reasons. For example, one user might have heard about a transaction because it is located near that transaction, while another user might be further away and the announcement of the transaction might not have made it across the network to that user yet. Or, there might be malicious users that have announced incorrect transactions and are trying to put wrong transactions into the blockchain.

Bitcoin uses a consensus protocol to work out which transactions are most likely correct and which ones are invalid. A consensus protocol is

essentially a way to tally up how many users have seen and believe in a transaction. In each round of the consensus protocol, new transactions are broadcast to all of the users in the Bitcoin network. These transactions must be signed by those spending the coins involved. Each user collects new transactions and puts the transactions it believes into a new block. In each round, a *random* user will get to announce its block to all of the other users. Bitcoin uses a clever way to determine which user will be this random user, as we shall soon discuss. All of the users that get this new block then check it over and, if they accept it, they include its hash in the next block that they create. Bitcoin uses the

SHA-256 hash function for hashing the block and forming the hash pointer.

Before we proceed to discuss how Bitcoin randomly chooses which user will be chosen to announce its block, we look at some security concerns with this protocol. First is the concern that Alice could attempt to double spend a coin. In each round, coins are being created and destroyed, and this is being logged into the ledger. Since the ledger is being announced to the broader community, other users know which coins have been spent previously and thus they will not accept a transaction where Alice tries to spend a coin she already used. Much of the trustworthiness of Bitcoin's system is derived from the fact that a consensus

protocol will arrive at what most users believe is correct and, as long as most users are honest, things will work as they should with false transactions being filtered before they get into the blockchain.

Another concern is that Alice could write transactions to give herself coins from someone else. This is not possible because, in order to do so, Alice would have to write a transaction as if she was another user, and this would require her to create a valid signature for another user, which she is unable to do.

A different concern is whether Alice could prevent a transaction from Bob from being added to the ledger. Although she could choose not to

include Bob's transaction in a block she is creating, there is a good chance that one of the other users in the network will announce a block with Bob's transaction because the user that is chosen to announce its block is randomly chosen. Additionally, even if she is chosen to announce her block, a different honest user will likely be chosen during the next round and will simply include Bob's transaction. In Bitcoin, there isn't any problem with a transaction taking a few rounds to get into the blockchain.

We now return to the question of how nodes are randomly selected. Remember that we needed to remove the role of a central entity like BigBank, and this

means that we cannot rely on a central entity to determine which user will be randomly selected. Bitcoin needed a way to choose a user randomly without anyone controlling who that user would be. At the same time, we can't just allow users to decide selfishly that they are the one chosen to announce the next block – that could lead to situations where a malicious user purposely leaves another user's transactions out of the blockchain.

Bitcoin also needed a way to encourage users to want to participate by being the random node that announces a new block. The design of Bitcoin allows users who create blocks to be given a reward for making a block. This reward

is known as a block reward, and is a fixed amount defined by the Bitcoin system. Additionally, when a user engages in a transaction that it wants logged into the blockchain ledger, it can set aside a small amount of value from the transaction (taking a little bit from the change it might issue itself, for example), and give this amount as a service charge to the user that creates and adds the block containing the transaction into the blockchain.

Together, these two incentives encourage users to want to make the block, so the challenge then is ensuring that random users are selected. Bitcoin achieves this by requiring that users perform a task where it isn't guaranteed

which user will be the first to complete the task. The first user to complete the task will have performed the *proof of work* needed to be randomly selected. Users compete with each other using their computing power, and the likelihood that a user will be selected ”randomly” depends on how much computing power they have. This means that if Alice has twice the computing power that Bob has, then she will be twice as likely to be the first to complete the computing task. But, even with this advantage, Alice does not control whether she will finish the task first, and so Bob and other users have a chance to publish the block before she does.

Bitcoin’s proof of work requires that

users solve a hash puzzle in order to be able to announce a block to be added to the blockchain. Specifically, a user who wishes to propose a block for adding the community's ledger is required to find a nonce such that

$$h(\text{Nonce}\|\text{prevhash}\|\text{TransX1}\|\text{TransX2}\|\cdots\|\text{TransXN})<B,$$

where Nonce is the nonce to be found, prevhash is the hash of the previous block in the blockchain, {TransXj} are the transactions that the user is proposing to add to the ledger, and B is a threshold value for the hash puzzle. What this means is the hash value is interpreted as an integer in binary and there must be a certain number of 0s at the beginning of this binary expansion. The value of B is chosen so that it will

take roughly T units of time for some user to solve the puzzle. In Bitcoin, T=10 minutes, and every two weeks the Bitcoin protocol adjusts B so that the average time needed to solve a hash puzzle remains around 10 minutes. Bitcoin uses the SHA-256 hashing algorithm twice for the proof of work cryptopuzzle, once for the hash of the previous ledger and once for obtaining the desired small hash value. In practice, the user does a massive computation using numerous nonces, hoping to be the first to find an appropriate hash value.

Once a user finds this nonce, it forms the new block, which includes Nonce, the previous hash, and the transactions it

has included. Then the user announces the block to the entire community. The reward for completing this task first is that the user will receive the block reward, which is several bitcoins, as well as transaction fees. The process of solving the hash puzzle and announcing a block is called Bitcoin mining, and has become a popular activity with many companies devoting large amounts of computing (and electrical power) to solving puzzles and reaping the Bitcoin rewards.

Let’s now put everything together and describe what a typical Bitcoin blockchain looks like. A block in Bitcoin’s blockchain consists of a header followed by a collection of

transactions that have been organized as a Merkle tree. The header itself contains several different pieces of information: the hash pointer to the previous block in the blockchain, a timestamp, the root of the Merkle tree, and the nonce. In actual implementation of Bitcoin, it is only the hash of the header that must satisfy the conditions of the cryptographic puzzle, rather than the hash of the full block of information. This makes it easier to verify a chain of blocks since one only needs to examine the hash of the headers, and not the hashes of many, many transactions. We may therefore visualize the Bitcoin blockchain as illustrated in Figure 16.1.

Figure 16.1 Each block in

Bitcoin's blockchain consists of a header and a Merkle tree of transactions

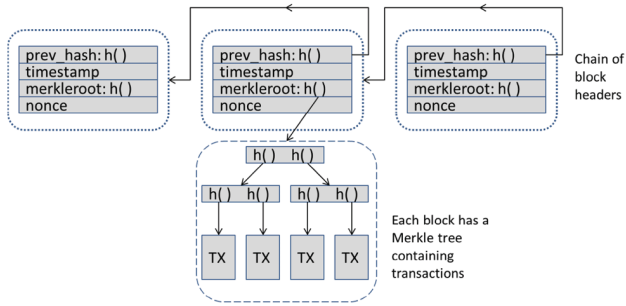


Figure 16.1 Full Alternative Text

Our description of Bitcoin is meant to convey the ideas behind cryptocurrencies in general and thus greatly simplifies the details of the actual Bitcoin protocol. We have avoided describing many practical matters, such as the formal specification of the language used by Bitcoin to specify transactions, the data structures and housekeeping that are needed to keep track of transactions, as well as the

details behind the use of the hash of a public key for addressing users. The reader can look at [Bitcoin] or [Narayanan et al.] to find the details behind Bitcoin and its protocols.

16.5 Exercises

1. In the scheme of Section 16.2, show that a valid coin satisfies the verification equations

   $gr \equiv ahH(A, B, z, a, b)$, $Ar \equiv zH(A, B, z, a, b, r)b$, and $g1r1g2r2 \equiv A$

2. In the scheme of Section 16.2, a hacker discovers the Bank's secret number x. Show how coins can be produced and spent without having an account at the bank.

3. In the scheme of Section 16.2, the numbers g1 and g2 are powers of g, but the exponents are supposed to be hard to find. Suppose we take g1=g2.

   1. Show that if the Spender replaces r1, r2 with r1′, r2′ such that r1+r2=r1′+r2′, then the verification equations still work.

   2. Show how the Spender can double spend without being identified.

4. Suppose that, in the scheme of Section 16.2, the coin is represented only as (A, B, a, r); for example, by ignoring z and b, taking the hash

function H to be a function of only A, B, a, and ignoring the verification equation $Ar \equiv zHb$. Show that the Spender can change the value of u to any desired number (without informing the Bank), compute a new value of I, and produce a coin that will pass the two remaining verification equations.

5. In the scheme of Section 16.2, if the Spender double spends, once with the Merchant and once with the Vendor, why is it very likely that $r2 - r2' \equiv 0 \pmod{q}$ (where r2, r2′ are as in the discussion of Fraud Control)?

6. A Sybil attack is one in which an entity claims multiple identities or roles in order to achieve an advantage against a system or protocol. Explain why there is no advantage in launching a Sybil attack against Bitcoin's proof of work approach to determining which user will randomly be selected to announce the next block in a blockchain.

7. Forgetful Fred would like to save a file containing his homework to a server in the network, which will allow him to download it later when he needs it. Describe an approach using hash functions that will allow Forgetful Fred to verify that he has obtained the correct file from the server, without requiring that Fred keep a copy of the entire file to check against the downloaded file.

8. A cryptographic hash puzzle involves finding a nonce n that, when combined with a message m, will hash to an output y that belongs to a target set S, i.e. $h(n\|x) \in S$.

    1. Assuming that all outputs of a k-bit cryptographic hash function are equally likely, find an expression for the average amount of nonces n that need to be tried in order to satisfy

       $h(n\|x)<2L$.

    2. Using your answer from part (a), estimate how many hashes it takes to obtain a hash value where the first 66 binary digits are 0s.

Chapter 17 Secret Sharing Schemes

Imagine, if you will, that you have made billions of dollars from Internet stocks and you wish to leave your estate to relatives. Your money is locked up in a safe whose combination only you know. You don't want to give the combination to each of your seven children because they are less than trustworthy. You would like to divide it among them in such a way that three of them have to get together to reconstruct the real combination. That way, someone who wants some of the inheritance must somehow cooperate with two other children. In this chapter we show how to

solve this type of problem.

17.1 Secret Splitting

The first situation that we present is the simplest. Consider the case where you have a message M, represented as an integer, that you would like to split between two people Alice and Bob in such a way that neither of them alone can reconstruct the message M. A solution to this problem readily lends itself: Give Alice a random integer r and give Bob M−r. In order to reconstruct the message M, Alice and Bob simply add their pieces together.

A few technical problems arise from the fact that it is impossible to choose a random integer in a way that all integers

are equally likely (the sum of the infinitely many equal probabilities, one for each integer, cannot equal 1). Therefore, we choose an integer n larger than all possible messages M that might occur and regard M and r as numbers mod n. Then there is no problem choosing r as a random integer mod n; simply assign each integer mod n the probability 1/n.

Now let us examine the case where we would like to split the secret among three people, Alice, Bob, and Charles. Using the previous idea, we choose two random numbers r and s mod n and give $M-r-s \pmod{n}$ to Alice, r to Bob, and s to Charles. To reconstruct the message M, Alice, Bob, and Charles simply add

their respective numbers.

For the more general case, if we wish to split the secret M among m people, then we must choose $m-1$ random numbers $r_1, \ldots, r_{m-1} \bmod n$ and give them to $m-1$ of the people, and $M - \sum_{k=1}^{m-1} r_k \pmod n$ to the remaining person.

17.2 Threshold Schemes

In the previous section, we showed how to split a secret among m people so that all m were needed in order to reconstruct the secret. In this section we present methods that allow a subset of the people to reconstruct the secret.

It has been reported that the control of nuclear weapons in Russia employed a safety mechanism where two out of three important people were needed in order to launch missiles. This idea is not uncommon. It's in fact a plot device that is often employed in spy movies. One can imagine a control panel with three slots for keys and the missile launch

protocol requiring that two of the three keys be inserted and turned at the same time in order to launch missiles to eradicate the earth.

Why not just use the secret splitting scheme of the previous section? Suppose some country is about to attack the enemy of the week, and the secret is split among three officials. A secret splitting method would need all three in order to reconstruct the key needed for the launch codes. This might not be possible; one of the three might be away on a diplomatic mission making peace with the previous week's opponent or might simply refuse because of a difference of opinion.

Definition

Let t, w be positive integers with $t \leq w$. A (t, w)**-threshold scheme** is a method of sharing a message M among a set of w participants such that any subset consisting of t participants can reconstruct the message M, but no subset of smaller size can reconstruct M.

The (t, w)-threshold schemes are key building blocks for more general sharing schemes, some of which will be explored in the Exercises for this chapter. We will describe two methods for constructing a (t, w)-threshold scheme.

The first method was invented in 1979 by Shamir and is known as the **Shamir**

threshold scheme or the Lagrange interpolation scheme. It is based upon some natural extensions of ideas that we learned in high school algebra, namely that two points are needed to determine a line, three points to determine a quadratic, and so on.

Choose a prime p, which must be larger than all possible messages and also larger than the number w of participants. All computations will be carried out mod p. The prime replaces the integer n of Section 17.1. If a composite number were to be used instead, the matrices we obtain might not have inverses.

The message M is represented as a number mod p, and we want to split it among w people in such a way that t of

them are needed to reconstruct the message. The first thing we do is randomly select t−1 integers mod p; call them s1, s2, ⋯ st−1. Then the polynomial

$$s(x) \equiv M+s1x+\cdots+st-1xt-1 \pmod{p}$$

is a polynomial such that $s(0) \equiv M \pmod{p}$. Now, for the w participants, we select distinct integers x1, …, xw (mod p) and give each person a pair (xi, yi) with $yi \equiv s(xi) \pmod{p}$. For example, 1, 2, …, w is a reasonable choice for the x's, so we give out the pairs (1, s(1)), …, (w, s(w)), one to each person. The prime p is known to all, but the polynomial s(x) is kept secret.

Now suppose t people get together and share their pairs. For simplicity of

notation, we assume the pairs are $(x1, y1), \cdots, (xt, yt)$. They want to recover the message M.

We begin with a linear system approach. Suppose we have a polynomial $s(x)$ of degree $t-1$ that we would like to reconstruct from the points $(x1, y1), \cdots, (xt, yt)$, where $yk=s(xk)$. This means that

$$yk \equiv M+s1xk1+\cdots+st-1xkt-1 \pmod p, \quad 1 \le k \le t.$$

If we denote $s0=M$, then we may rewrite this as

$$1x1\cdots x1t-11x2\cdots x2t-1\vdots\ddots\vdots1xt\cdots xtt-1s0s1\vdots st-1 \equiv y1y2\vdots yt \pmod p.$$

The matrix, let's call it V, is what is known as a Vandermonde matrix. We know that this system has a unique solution mod p if the determinant of the

matrix V is nonzero mod p (see Section 3.8). It can be shown that the determinant is

$$\det V = \prod 1 \le j < k \le t (x_k - x_j),$$

which is zero mod p only when two of the xi's coincide mod p (this is where we need p to be prime; see Exercise 13(a) in Chapter 3). Thus, as long as we have distinct xk's, the system has a unique solution.

We now describe an alternative approach that leads to a formula for the reconstruction of the polynomial and hence for the secret message. Our goal is to reconstruct a polynomial s(x) given that we know t of its values (xk, yk). First, let

$$l_k(x) = \prod_{i=1, i \neq k}^{t} \frac{x - x_i}{x_k - x_i} \pmod{p}.$$

Here, we work with fractions mod p as described in Section 3.3. Then

$$l_k(x_j) \equiv \begin{cases} 1 & \text{when } k = j \\ 0 & \text{when } k \neq j. \end{cases}$$

This is because $l_k(x_k)$ is a product of factors $(x_k - x_i)/(x_k - x_i)$, all of which are 1. When $k \neq j$, the product for $l_k(x_j)$ contains the factor $(x_j - x_j)/(x_k - x_j)$, which is 0.

The **Lagrange interpolation polynomial**

$$p(x) = \sum_{k=1}^{t} y_k l_k(x)$$

satisfies the requirement $p(x_j) = y_j$ for $1 \leq j \leq t$. For example,

$$p(x_1) = y_1 l_1(x_1) + y_2 l_2(x_2) + \cdots \equiv y_1 \cdot 1 + y_2 \cdot 0 + \cdots \equiv y_1 \pmod{p}.$$

We know from the previous approach

with the Vandermonde matrix that the polynomial s(x) is the only one of degree t−1 that takes on the specified values. Therefore, p(x)=s(x).

Now, to reconstruct the secret message all we have to do is calculate p(x) and evaluate it at x=0. This gives us the formula

$$M \equiv \sum_{k=1}^{t} y_k \prod_{j=1, j \neq k}^{t} \frac{-x_j}{x_k - x_j} \pmod{p}.$$

Example

Let's construct a (3, 8)-threshold scheme. We have eight people and we want any three to be able to determine the secret, while two people cannot determine any information about the message.

Suppose the secret is the number M=190503180520 (which corresponds to the word *secret*). Choose a prime p, for example, p=1234567890133 (we need a prime at least as large as the secret, but there is no advantage in using primes much larger than the maximum size of the secret). Choose random numbers s1 and s2 mod p and form the polynomial

$$s(x)=M+s1x+s2x2.$$

For example, let's work with

$$s(x)=190503180520+482943028839x+1206749628665x2.$$

We now give the eight people pairs (x, s(x)). There is no need to choose the values of x randomly, so we simply use x=1, 2, …, 8. Therefore, we distribute the following pairs, one to each person:

(1, 645627947891)(2, 1045116192326)(3, 154400023692)
(4, 442615222255)(5, 675193897882)(6, 852136050573)
(7, 973441680328)(8, 1039110787147).

Suppose persons 2, 3, and 7 want to collaborate to determine the secret. Let's use the Lagrange interpolating polynomial. They calculate that the following polynomial passes through their three points:

20705602144728/5−1986192751427x+(1095476582793/5)x2.

At this point they realize that they should have been working mod p. But

740740734080×5 ≡ 1 (mod p),

so they replace 1/5 by 740740734080, as in Section 3.3, and reduce mod p to obtain

190503180520+482943028839x+1206749628665x2.

This is, of course, the original

polynomial s(x). All they care about is the constant term 190503180520, which is the secret. (The last part of the preceding calculations could have been shortened slightly, since they only needed the constant term, not the whole polynomial.)

Similarly, any three people could reconstruct the polynomial and obtain the secret.

If persons 2, 3, and 7 chose the linear system approach instead, they would need to solve the following:

$$1241391749Ms1s2 \equiv 104511619232615440002369297344168032 8 \pmod{1234567890133}.$$

This yields

$$(M, s1, s2) \equiv (190503180520, 482943028839, 1206749628665),$$

so they again recover the polynomial and the message.

What happens if only two people get together? Do they obtain any information? For example, suppose that person 4 and person 6 share their points (4, 442615222255) and (6, 852136050573) with each other. Let c be any possible secret. There is a unique quadratic polynomial ax2+bx+c passing through the points (0, c), (4, 442615222255), and (6, 852136050573). Therefore, any secret can still occur.

Similarly, they cannot guess the share held, for example, by person 7: Any point (7, y7) yields a unique secret c, and any secret c yields a polynomial

ax2+bx+c, which corresponds to y7=49a +7b+c. Therefore, every value of y7 can occur, and each corresponds to a secret. So persons 4 and 6 don't obtain any additional information about which secrets are more likely when they have only their own two points.

Similarly, if we use a polynomial of degree t−1, there is no way that t−1 persons can obtain information about the message with only their data. Therefore, t people are required to obtain the message.

For another example, see Example 38 in the Computer Appendices.

There are other methods that can be used for secret sharing. We now

describe one due to Blakley, also from 1979. Suppose there are several people and we want to arrange that any three can find the secret, but no two can. Choose a prime p and let x0 be the secret. Choose y0, z0 randomly mod p. We therefore have a point Q=(x0, y0, z0) in three-dimensional space mod p. Each person is given the equation of a plane passing through Q. This is accomplished as follows. Choose a, b randomly mod p and then set $c \equiv z0 - ax0 - by0 \pmod{p}$. The plane is then

$$z=ax+by+c.$$

This is done for each person. Usually, three planes will intersect in a point, which must be Q. Two planes will

intersect in a line, so usually no information can obtained concerning the secret x0 (but see Exercise 13).

Note that only one coordinate should be used to carry the secret. If the secret had instead been distributed among all three coordinates x0, y0, z0, then there might be only one meaningful message corresponding to a point on a line that is the intersection of two persons' planes.

The three persons who want to deduce the secret can proceed as follows. They have three equations

$$aix+biy-z \equiv -ci \pmod{p}, \quad 1 \le i \le 3,$$

which yield the matrix equation

$$a1b1-1a2b2-1a3b3-1x0y0z0 \equiv -c1-c2-c3.$$

As long as the determinant of this matrix is nonzero mod p, the matrix can be inverted mod p and the secret x0 can be found (of course, in practice, one would tend to solve this by row operations rather than by inverting the matrix).

Example

Let p=73. Suppose we give A, B, C, D, E the following planes:

A: z=4x+19y+68B: z=52x+27y+10C: z=36x+65y+18D: z=57x+12y+16E: z=34x+19y+49.

If A, B, C want to recover the secret, they solve

$$\begin{pmatrix} 4 & 19 & -1 \\ 52 & 27 & -1 \\ 36 & 65 & -1 \end{pmatrix} \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} \equiv \begin{pmatrix} -68 \\ -10 \\ -18 \end{pmatrix} \pmod{73}.$$

The solution is (x0, y0, z0)=(42, 29, 57), so the secret is x0=42. Similarly, any

three of A, B, C, D, E can cooperate to recover x0.

By using (t−1)-dimensional hyperplanes in t-dimensional space, we can use the same method to create a (t, w)-threshold scheme for any values of t and w.

As long as p is reasonably large, it is very likely that the matrix is invertible, though this is not guaranteed. It would not be hard to arrange ways to choose a, b, c so that the matrix is always invertible. Essentially, this is what happens in the Shamir method. The matrix equations for both methods are similar, and the Shamir method could be regarded as a special case of the Blakley method. But since the Shamir method yields a Vandermonde matrix, the

equations can always be solved. The other advantage of the Shamir method is that it requires less information to be carried by each person: (x, y) versus (a, b, c, …).

We now return to the Shamir method and consider variations of the basic situation. By giving certain persons more shares, it is possible to make some people more important than others. For example, suppose we have a system in which eight shares are required to obtain the secret, and suppose the boss is given four shares, her daughters are given two shares, and the other employees are each given one share. Then the boss and two of her daughters can obtain the secret, or three daughters and two regular

employees, for example.

Here is a more complicated situation. Suppose two companies A and B share a bank vault. They want a system where four employees from A and three from B are needed in order to obtain the secret combination. Clearly it won't work if we simply supply shares that are all for the same secret, since one company could simply use enough partial secrets from its employees that the other company's shares would not be needed. The following is a solution that works. Write the secret s as the sum of two numbers $s \equiv cA+cB \pmod{p}$. Now make cA into a secret shared among the employees of A as the constant term of a polynomial of degree

3. Similarly, let cB be the constant term of a polynomial of degree 2 and use this to distribute shares of cB among the employees of B. If four employees of A and three employees of B get together, then those from A determine cA and those from B determine cB. Then they add cA and cB to get s.

Note that A does not obtain any information about the secret s by itself since $cA+x \equiv s \pmod{p}$ has a unique solution x for every s, so every possible value of s corresponds to a possible value of cB. Therefore, knowing cA does not help A to find the secret; A also needs to know cB.

17.3 Exercises

1. Suppose you have a secret, namely 5. You want to set up a system where four persons A, B, C, D are given shares of the secret in such a way that any two of them can determine the secret, but no one alone can determine the secret. Describe how this can be done. In particular, list the actual pieces of information (i.e., numbers) that you could give to each person to accomplish this.

2. Persons A, B, C participate in a Shamir (3, 2) secret sharing scheme. They work mod 11. A receives the share (1, 5), B receives (2, 9), and C receives (3, 3).

    1. Show that at least one of the three shares is incorrect.

    2. Suppose A and C have correct shares. Find the secret.

3. You set up a (2, 30) Shamir threshold scheme, working mod the prime 101. Two of the shares are (1,13) and (3,12). Another person received the share (2, *), but the part denoted by * is

unreadable. What is the correct value of * ?

4. You set up a (2, 10) Shamir threshold scheme, working mod the prime 73. Two of the shares are (1, 10) and (2, 18). A third share is (5, *). What is *?

5. In a (3, 5) Shamir secret sharing scheme with modulus p=17, the following were given to Alice, Bob, and Charles: (1, 8), (3, 10), (5, 11). Calculate the corresponding Lagrange interpolating polynomial, and identify the secret.

6. In a Shamir secret sharing scheme, the secret s is the constant term of a degree 4 polynomial mod the prime 1093. Suppose three people have the secrets (2, 197), (4, 874), and (13, 547). How many possibilities are there for the secret? (*Note:* We assume that $0 \leq s \leq 1092$.)

7. Mark doesn't like mods, so he wants to implement a (2, 30) Shamir secret sharing scheme without them. His secret is M (a positive integer) and he gives person i the share (i, M+si) for a positive integer s that he randomly chooses. Bob receives the share (20, 97). Describe how Bob can narrow down the possibilities for M and determine what values of M are possible.

8. A key distributor uses a (2, 20)-threshold scheme

to distribute a combination to an electronic safe to 20 participants.

    1. What is the smallest number of participants needed to open the safe, given that one unknown participant is a cheater who will reveal a random share?

    2. If they are only allowed to try one combination (if they are wrong the electronic safe shuts down permanently), then how many participants are necessary to open the safe? (Note: This one is a little subtle. A majority vote actually works with four people, but you need to show that a tie cannot occur.)

9. A certain military office consists of one general, two colonels, and five desk clerks. They have control of a powerful missile but don't want the missile launched unless the general decides to launch it, or the two colonels decide to launch it, or the five desk clerks decide to launch it, or one colonel and three desk clerks decide to launch it. Describe how you would do this with a secret sharing scheme. (Hint: Try distributing the shares of a (10, 30) Shamir scheme.)

10. Suppose there are four people in a room, exactly

one of whom is a foreign agent. The other three people have been given pairs corresponding to a Shamir secret sharing scheme in which any two people can determine the secret. The foreign agent has randomly chosen a pair. The people and pairs are as follows. All the numbers are mod 11.

A:(1, 4) B:(3, 7) C:(5, 1) D:(7, 2).

Determine who the foreign agent is and what the message is.

11. Consider the following situation: Government A, Government B, and Government C are hostile to each other, but the common threat of Antarctica looms over them. They each send a delegation consisting of 10 members to an international summit to consider the threat that Antarctica's penguins pose to world security. They decide to keep a watchful eye on their tuxedoed rivals. However, they also decide that if the birds get too rowdy, then they will launch a full-force attack on Antarctica. Using secret sharing techniques, describe how they can arrange to share the launch codes so that it is necessary that three members from delegation A, four members from delegation B, and two members from C cooperate to reconstruct the launch codes.

12. This problem explores what is known as the

Newton form of the interpolant. In the Shamir method, we presented two methods for calculating the interpolating polynomial. The system of equations approach is difficult to solve and easy to evaluate, while with the Lagrange approach it is quite simple to determine the interpolating polynomial but becomes a labor to evaluate. The Newton form of the interpolating polynomial comes from choosing $1, x-x1, (x-x1)(x-x2), \cdots, (x-x1)(x-x2)\cdots(x-xt)$ as a basis. The interpolating polynomial is then $p(x)=c0+c1(x-x1)+c2(x-x1)(x-x2)+\cdots+ct(x-x1)(x-x2)\cdots(x-xt)$. Show that we can solve for the coefficients ck by solving a system $Nc=y$. What special properties do you observe in the matrix N? Why does this make the system easier to solve?

13. In a Blakley (3, w) scheme, suppose persons A and B are given the planes $z=2x+3y+13$ and $z=5x+3y+1$. Show that they can recover the secret without a third person.

17.4 Computer Problems

1. Alice, Bob, and Charles have each received shares of a secret that was split using the secret splitting scheme described in Section 17.1. Suppose that $n=2110763$. Alice is given the share $M-r-s=1008369$, Bob is given the share $r=593647$, and Charles is given the share $s=631870$. Determine the secret M.

2. For a Shamir (4,7) secret sharing scheme, take $p=8737$ and let the shares be

   $$(1,\ 214),\ (2,\ 7543),\ (3,\ 6912),\ (4,\ 8223),\ (5,\ 3904),\ (6,\ 3857),$$

   Take a set of four shares and find the secret. Now take another set of four shares and verify that the secret obtained is the same.

3. Alice, Bob, Charles, and Dorothy use a (2, 4) Shamir secret sharing scheme using the prime $p=984583$. Suppose that Alice gets the share (38, 358910), Bob gets the share (3876, 9612), Charles gets the share (23112, 28774), and Dorothy gets the share (432, 178067). One of these shares was incorrectly received. Determine which one is

incorrect, and find the secret.

Chapter 18 Games

18.1 Flipping Coins over the Telephone

Alice is living in Anchorage and Bob is living in Baltimore. A friend, not realizing that they are no longer together, leaves them a car in his will. How do they decide who gets the car? Bob phones Alice and says he'll flip a coin. Alice chooses "Tails" but Bob says "Sorry, it was Heads." So Bob gets the car.

For some reason, Alice suspects Bob might not have been honest. (Actually, he told the truth; as soon as she called

tails, he pulled out his specially made two-headed penny so he wouldn't have to lie.) She resolves that the next time this happens, she'll use a different method. So she goes to her local cryptologist, who suggests the following method.

Alice chooses two large random primes p and q, both congruent to 3 mod 4. She keeps them secret but sends the product n=pq to Bob. Then Bob chooses a random integer x and computes $y \equiv x2 \pmod{n}$. He keeps x secret but sends y to Alice. Alice knows that y has a square root mod n (if it doesn't, her calculations will reveal this fact, in which case she accuses Bob of cheating), so she uses her knowledge of

p and q to find the four square roots ±a, ±b of y (mod n) (see Section 3.9). One of these will be x, but she doesn't know which one. She chooses one at random (this is the "flip"), say b, and sends it to Bob. If $b \equiv \pm x \pmod{n}$, Bob tells Alice that she wins. If $b \equiv \pm x \pmod{n}$, Bob wins.

AliceBobn=pq→ny←y ≡ x2a2 ≡ b2 ≡ y→bAlice winsorBob wins←←b ≡ ±xorb ≡ ±x

But, asks Alice, how can I be sure Bob doesn't cheat? If Alice sends b to Bob and $x \equiv \pm a \pmod{n}$, then Bob knows all four square roots of y (mod n), so he can factor n. In particular, gcd(x−b, n) gives a nontrivial factor of n. Therefore, if it is computationally infeasible to factor n, the only way Bob could produce the factors p and q would be when his value

of x is not plus or minus the value of b that Alice sends. If Alice sends Bob ±x, Bob has no more information than he had when Alice sent him the number n. Therefore, he should not be able to produce p and q in this case. So Alice can check that Bob didn't cheat by asking Bob for the factorization of n.

What if Alice tries to cheat by sending Bob a random number rather than a square root of y? This would surely prevent Bob from factoring n. Bob can guard against this by checking that the square of the number Alice sends is congruent to y.

Suppose Alice tries to deceive Bob by sending a product of three primes. Of course, Bob could ask Alice for the

factorization of n at the end of the game; if Alice produces two factors, they can be quickly checked for primality. But Bob shouldn't worry about this possibility. When n is the product of three distinct primes, there are eight square roots of y. Therefore, up to sign there are four choices of numbers for Alice to send. Each of the three wrong choices will allow Bob to find a nontrivial factor of n. So Alice would decrease her chances of winning to only one in four. Therefore, she should not try this.

There is one flaw in this procedure. Suppose Bob decides he wants to lose. He can then claim his value of x was exactly the value that Alice sent him.

Alice cannot dispute this since the only information she has is the square of Bob's number, which is congruent to the square of her number. There are other procedures that can prevent Bob from trying to lose, but we will not discuss them here.

Finally, we should mention that it is not difficult to find primes p and q that are congruent to 3 mod 4. The density of primes congruent to 1 mod 4 is the same as the density of primes that are 3 mod 4. Therefore, find a random prime p. If it is not 3 mod 4, try another. This process should succeed quickly. We can find q similarly.

Example

Alice chooses

p=2038074743 and q=1190494759.

She sends

n=pq=2426317299991771937

to Bob. Bob takes

x=1414213562373095048

(this isn't as random as it looks; but Bob thinks the decimal expansions of square roots look random) and computes

y ≡ x2 ≡ 363278601055491705 (mod n),

which he sends to Alice.

Alice computes

y(p+1)/4 ≡ 1701899961 (mod p) and y(q+1)/4 ≡ 325656728 (mod q).

Therefore, she knows that

$x \equiv \pm 1701899961 \pmod{p}$ and $x \equiv \pm 325656728 \pmod{q}$.

The Chinese remainder theorem puts these together in four ways to yield

$x \equiv \pm 1012103737618676889$ or $\pm 937850352623334103 \pmod{n}$.

Suppose Alice sends 1012103737618676889 to Bob. This is $-x \pmod{n}$, so Bob declares Alice the winner.

Suppose instead that Alice sends 9378503526233341O3 to Bob. Then Bob claims victory. By computing

gcd(1414213562373095048 -
937850352623334103, n) = 1190494759,

he can prove that he won.

18.2 Poker over the Telephone

Alice and Bob quickly tire of flipping coins over the telephone and decide to try poker. Bob pulls out his deck of cards, shuffles, and deals two hands, one for Alice and one for himself. Now what does he do? Alice won't let him read the cards to her. Also, she suggests that he might not be playing with a full deck. Arguments ensue. But then someone suggests that they each choose their own cards. The betting is fast and furious. After several hundred coins (they remain unused from the coin-flipping protocol) have been wagered, Alice and Bob discover that they each have a royal

flush. Each claims the other must have cheated. Fortunately, their favorite cryptologist can help.

Here is the method she suggests, in nonmathematical terms. Bob takes 52 identical boxes, puts a card in each box, and puts a lock on each one. He dumps the boxes in a bag and sends them to Alice. She chooses five boxes, puts her locks on them, and sends them back to Bob. He takes his locks off and sends the five boxes back to Alice, who takes her locks off and finds her five cards. Then she chooses five more boxes and sends them back to Bob. He takes off his locks and gets his five cards. Now suppose Alice wants to replace three cards. She puts three cards in a discard

box, puts on her lock, and sends the box to Bob. She then chooses three boxes from the remaining 42 card boxes, puts on her locks, and sends them to Bob. Bob removes his locks and sends them back to Alice, who removes her locks and gets the cards. If Bob wants to replace two cards, he puts them in another discard box, puts on his lock, and sends the box to Alice. She chooses two card boxes and sends them to Bob. He removes his locks and gets his cards. They then compare hands to see who wins. We'll assume Alice wins.

After the hand has been played, Bob wants to check that Alice put three cards in her discard box since he wants to be sure she wasn't playing with eight cards.

He puts his lock on the box and sends the box to Alice, who takes her lock off. Since Bob's lock is still on the box, she can't change the contents. She sends the box back to Bob, who removes the lock and finds the three cards that Alice discarded (this differs from standard poker in that Bob sees the actual cards discarded; in a standard game, Bob only sees that Alice discards three cards and doesn't need to look at them afterward). Similarly, Alice can check that Bob discarded two cards.

Bob can check that Alice played with the hand that was dealt by asking her to send her cards to him. Alice cannot change her hand since all the remaining cards still have Bob's locks on them

(and Bob can't open them since Alice has them in her possession).

Of course, various problems arise if Alice or Bob unjustly accuses the other of cheating. But, ignoring such complications, we see that Alice and Bob can now play poker. However, the postage for sending 52 boxes back and forth is starting to cut into Alice's profits. So she goes back to her cryptologist and asks for a mathematical implementation. The following is the method.

Alice and Bob agree on a large prime p. Alice chooses a secret integer α with $\gcd(\alpha, p-1)=1$, and Bob chooses a secret integer β with $\gcd(\beta, p-1)=1$. Alice computes α' such that $\alpha\alpha' \equiv 1 \pmod{p-1}$

and Bob computes β' with $\beta\beta' \equiv 1 \pmod{p-1}$. A different α and β are used for each hand. A different p could be used for each hand also.

Note that $c\alpha\alpha' \equiv c \pmod{p}$, and similarly for β. This can be seen as follows: $\alpha\alpha' \equiv 1 \pmod{p-1}$, so $\alpha\alpha'=1+(p-1)k$ for some integer k. Therefore, when $c \equiv 0 \pmod{p}$

$$c\alpha\alpha' \equiv c\cdot(cp-1)k \equiv c\cdot 1k \equiv c \pmod{p}.$$

Trivially, we also have $c\alpha\alpha' \equiv c \pmod{p}$ when $c \equiv 0 \pmod{p}$.

The 52 cards are changed to 52 distinct numbers $c1, \ldots, c52 \bmod p$ via some prearranged scheme. Bob computes $bi \equiv ci\beta \pmod{p}$ for $1 \leq i \leq 52$, randomly permutes these numbers, and sends them

to Alice. Alice chooses five numbers $b_{i1}, \ldots, b_{i5}$, computes $b_{ij}^{\alpha} \pmod{p}$ for $1 \le j \le 5$, and sends these numbers to Bob. Bob takes off his lock by raising these numbers to the β' power and sends them to Alice, who removes her lock by raising to the α' power. This gives Alice her hand.

Alice then chooses five more of the numbers b_i and sends them back to Bob, who removes his locks by raising the numbers to the β' power. This gives him his hand. The rest of the game proceeds in this fashion.

It seems to be quite difficult for Alice to deduce Bob's cards. She could guess which encrypted card b_i corresponds to a fixed unencrypted card c_j. This means

Alice would need to solve equations of the form $cj\beta \equiv bi \pmod{p}$ for β. Doing this for the 52 choices for bi would give at most 52 choices for β. The correct exponent β could then be determined by choosing another card cj' and trying the various possibilities for β to see which ones give the encrypted values that are on the list of encrypted cards. But these equations that Alice needs to solve are discrete logarithm problems, which are generally assumed to be difficult when p is large (see Chapter 10).

Example

Let's consider a simplified game where there are only five cards: ten, jack, queen, king, ace. Each player is dealt one card. The winner is the one with the

higher card. Change the cards to numbers using a=01, b=02, …, so we have the following:

TenJackQueenKingAce2005141001031117210505141109140710305

Let the prime be p=2396271991. Alice chooses her secret α=1234567 and Bob chooses his secret β=7654321. Alice computes α'=402406273 and Bob computes β'=200508901. This can be done via the extended Euclidean algorithm. Just to be sure, Alice checks that $\alpha\alpha' \equiv 1 \pmod{p-1}$, and Bob does a similar calculation with β and β'.

Bob now calculates (congruences are mod p)

200514β ≡ 91401222410010311β ≡ 150729877017210505 14β ≡ 74390103110

He shuffles these numbers and sends

them to Alice:

1507298770, 1112225809, 2337996540, 914012224, 74390103.

Since Alice does not know β, it is unlikely she can deduce which card is which without a lot of computation.

Alice now chooses her card by choosing one of these numbers – for example, the fourth – raises it to the power α, and sends it to Bob:

$914012224^{\alpha} \equiv 1230896099 \pmod{p}$.

Bob takes off his lock by raising this to the power β' and sends it back to Alice:

$1230896099^{\beta'} \equiv 1700536007 \pmod{p}$.

Alice now removes her lock by raising this to the power α':

$1700536007\alpha' \equiv 200514 \pmod{p}$.

Her card is therefore the ten.

Now Alice chooses Bob's card by simply choosing one of the original cards she received – for example, 1507298770 – and sending it back to Bob. Bob computes

$1507298770\beta' \equiv 10010311 \pmod{p}$.

Therefore, his card is the jack.

This accomplishes the desired dealing of the cards. Alice and Bob now compare cards and Bob wins. To prevent cheating, Alice and Bob then reveal their secret exponents α and β. Suppose Alice tries to claim she has the king. Bob can quickly compute α' and show that the card he sent to Alice was the

ten.

For another example of this game, see Example 39 in the Computer Appendices.

18.2.1 How to Cheat

No game of poker would be complete without at least the possibility of cheating. Here's how to do it in the present situation.

Bob goes to his local number theorist, who tells him about quadratic residues. A number r (mod p) is called a **quadratic residue** mod p if the congruence $x^2 \equiv r \pmod{p}$ has a solution; in other words, r is a square mod p. A nonresidue n is an integer

such that $x2 \equiv n \pmod{p}$ has no solution.

There is an easy way to decide whether or not a number $z \equiv 0 \pmod{p}$ is a quadratic residue or nonresidue:

$z^{(p-1)/2} \equiv +1 \pmod{p}$ if z is a quadratic residue $-1 \pmod{p}$ if z is a quadratic nonresidue

(see Exercise 1). This determination can also be done using the Legendre or Jacobi symbol plus quadratic reciprocity. See Section 3.10.

Recall that we needed $\gcd(\alpha, p-1)=1$ and $\gcd(\beta, p-1)=1$. Therefore, α and β are odd. A card c is encrypted to $c\beta$, and

$(c\beta)^{(p-1)/2} \equiv (c^{(p-1)/2})\beta \equiv c^{(p-1)/2} \pmod{p}$,

since $(\pm 1)^{\text{odd}} \equiv \pm 1$ (with the same choice of signs on both sides of the

congruence). Therefore, c is a quadratic residue mod p if and only if cβ is a quadratic residue. The corresponding statement also applies to the α and αβ power of the cards.

When Alice sends Bob the five cards that will make up her hand, Bob quickly checks these cards to see which are quadratic residues and which are nonresidues. This means that there are two sets R and N, and for each of Alice's cards, he knows whether the card is in R or N. This gives him a slight advantage. For example, suppose he needs to know whether or not she has the queen of hearts and he determines that it is in N. If she has only one N card, the chances are low that she has

the card. In this way, Bob obtains a slight advantage and starts winning.

Alice quickly consults her local cryptologist, who fortunately knows about quadratic residues, too. Now when Alice chooses Bob's hand, she arranges that all of his cards are in R, for example. Then she knows that his hand is chosen from 26 cards rather than 52. This is better than the partial information that Bob has and is useful enough that she gains an advantage over Bob. Finally, Alice gets very bold. She sneakily chooses the prime p so that the ace, king, queen, jack, and ten of spades are the only quadratic residues. When she chooses Bob's hand, she gives him five nonresidues. She chooses the five

residues for herself. Bob, who has been computing residues and nonresidues on each hand, has already been getting suspicious since his cards have all been residues or all been nonresidues for several hands. But now he sees before the hand is played that she has chosen a royal flush for herself. He accuses her of cheating, arguments ensue, and they go back to coin flipping.

Example

Let's return to the simplified example. The choice of prime p was not random. In fact,

$$200514^{(p-1)/2} \equiv -1,\ 10010311^{(p-1)/2} \equiv -1,\ 1721050514^{(p-1)/2} \equiv -1,\ 11091407^{(p-1)/2} \equiv -1,\ 10305^{(p-1)/2} \equiv -1,$$

so only the ace is a nonresidue, while all

the remaining cards are quadratic residues.

When Alice is choosing her hand, she computes

$$1507298770^{(p-1)/2} \equiv -1, 1112225809^{(p-1)/2} \equiv -1, 2337996540^{(p-1)/2} \equiv -1, 914012224^{(p-1)/2} \equiv -1, 74390103^{(p-1)/2} \equiv -1.$$

This tells her that the ace is 1112225809. She raises it to the power α', then sends it to Bob. He raises it to the power β' and sends it back to Alice, who raises it to the power α'. Of course, she finds that her card is the ace.

For more on playing poker over the telephone, see [Fortune-Merritt].

18.3 Exercises

1. Let α be a primitive root for the prime p. This means that the numbers $1, \alpha, \alpha 2, \alpha 3, \ldots, \alpha p-2 \pmod{p}$ yield all of the nonzero congruence classes mod p.

    1. Let i be fixed and suppose $x2 \equiv \alpha i \pmod{p}$ has a solution x. Show that i must be even. (Hint: Write $x \equiv \alpha j$ for some j. Now use the fact that $\alpha k \equiv \alpha l \pmod{p}$ if and only if $k \equiv l \pmod{p-1}$.) This shows that the nonzero squares mod p are exactly $1, \alpha 2, \alpha 4, \alpha 6, \ldots \pmod{p}$, and therefore $\alpha, \alpha 3, \alpha 5, \ldots$ are the quadratic nonresidues mod p.

    2. Using the definition of primitive root, show that $\alpha(p-1)/2 \equiv 1 \pmod{p}$.

    3. Use Exercise 15 in Chapter 3 to show that $\alpha(p-1)/2 \equiv -1 \pmod{p}$.

    4. Let $x \equiv 0 \pmod{p}$. Show that $x(p-1)/2 \equiv 1 \pmod{p}$ if x is a quadratic residue and $x(p-1)/2 \equiv -1 \pmod{p}$ if x is

a quadratic nonresidue mod p.

2. In the coin flipping protocol with n=pq, suppose Bob sends a number y such that neither y nor −y has a square root mod n.

    1. Show that y cannot be a square both mod p and mod q. Similarly, −y cannot be a square mod both primes.

    2. Suppose y is not a square mod q. Show that −y is a square mod q.

    3. Show that y is a square mod one of the primes and −y is a square mod the other.

    4. Benevolent Alice decides to correct Bob's "mistake." Suppose y is a square mod p and −y is a square mod q. Alice calculates a number b such that $b2 \equiv y \pmod{p}$ and $b2 \equiv -y \pmod{q}$ and sends b to Bob (there are two pairs of choices for b). Show how Bob can use this information to factor n and hence claim victory.

3.

    1. Let p be an odd prime. Show that if $x \equiv -x \pmod{p}$, then $x \equiv 0 \pmod{p}$.

2. Let p be an odd prime. Suppose x, y $\equiv 0 \pmod{p}$ and $x^2 \equiv y^2 \pmod{p^2}$. Show that $x \equiv \pm y \pmod{p^2}$ (Hint: Look at the proof of the Basic Principle in Section 9.3.)

3. Suppose Alice cheats when flipping coins by choosing p=q. Show that Bob always loses in the sense that Alice always returns ±x. Therefore, it is wise for Bob to ask for the two primes at the end of the game.

Chapter 19 Zero-Knowledge Techniques

19.1 The Basic Setup

A few years ago, it was reported that some thieves set up a fake automatic teller machine at a shopping mall. When a person inserted a bank card and typed in an identification number, the machine recorded the information but responded with the message that it could not accept the card. The thieves then made counterfeit bank cards and went to legitimate teller machines and withdrew cash, using the identification numbers they had obtained.

How can this be avoided? There are several situations where someone reveals a secret identification number or password in order to complete a transaction. Anyone who obtains this secret number, plus some (almost public) identification information (for example, the information on a bank card), can masquerade as this person. What is needed is a way to use the secret number without giving any information that can be reused by an eavesdropper. This is where zero-knowledge techniques come in.

The basic challenge-response protocol is best illustrated by an example due to Quisquater, Guillou, and Berson [Quisquater et al.]. Suppose there is a

tunnel with a door, as in Figure 19.1. Peggy (the prover) wants to prove to Victor (the verifier) that she can go through the door without giving any information to Victor about how she does it. She doesn't even want to let Victor know which direction she can pass through the door (otherwise, she could simply walk down one side and emerge from the other). They proceed as follows. Peggy enters the tunnel and goes down either the left side or the right side of the tunnel. Victor waits outside for a minute, then comes in and stands at point B. He calls out "Left" or "Right" to Peggy. Peggy then comes to point B by the left or right tunnel, as requested. This entire protocol is repeated several times, until Victor is

satisfied. In each round, Peggy chooses which side she will go down, and Victor randomly chooses which side he will request.

Figure 19.1 The Tunnel Used in the Zero-Knowledge Protocol

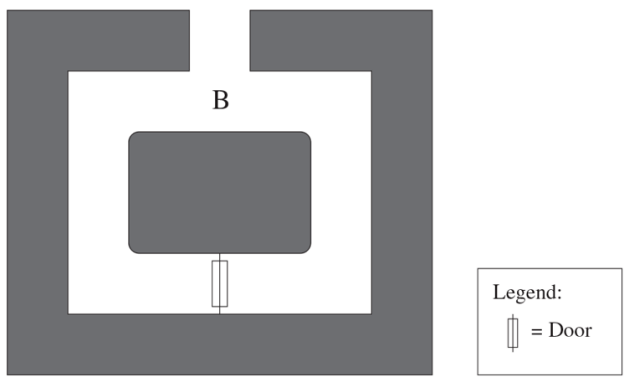


Figure 19.1 Full Alternative Text

Since Peggy must choose to go down the left or right side before she knows what Victor will say, she has only a 50% chance of fooling Victor if she

doesn’t know how to go through the door. Therefore, if Peggy comes out the correct side for each of 10 repetitions, there is only one chance in 210=1024 that Peggy doesn’t know how to go through the door. At this point, Victor is probably convinced, though he could try a few more times just to be sure.

Suppose Eve is watching the proceedings on a video monitor carried by Victor. She will not be able to use anything she sees to convince Victor or anyone else that she, too, can go through the door. Moreover, she might not even be convinced that Peggy can go through the door. After all, Peggy and Victor could have planned the sequence of rights and lefts ahead of time. By this

reasoning, there is no useful information that Victor obtains that can be transmitted to anyone.

Note that there is never a proof, in a strict mathematical sense, that Peggy can go through the door. But there is overwhelming evidence, obtained through a series of challenges and responses. This is a feature of zero-knowledge "proofs."

There are several mathematical versions of this procedure, but we'll concentrate on one of them. Let n=pq be the product of two large primes. Let y be a square mod n with gcd(y, n)=1. Recall that finding square roots mod n is hard; in fact, finding square roots mod n is equivalent to factoring n (see Section

3.9). However, Peggy claims to know a square root s of y. Victor wants to verify this, but Peggy does not want to reveal s. Here is the method:

1. Peggy chooses a random number r1 and lets $r2 \equiv sr1-1 \pmod{n}$, so

   $r1r2 \equiv s \pmod{n}$.

   (We may assume that gcd(r1, n)=1, so $r1-1 \pmod{n}$ exists; otherwise, Peggy has factored n.) She computes

   $x1 \equiv r12, x2 \equiv r22 \pmod{n}$

   and sends x1 and x2 to Victor.

2. Victor checks that $x1x2 \equiv y \pmod{n}$, then chooses either x1 or x2 and asks Peggy to supply a square root of it. He checks that it is an actual square root.

3. The first two steps are repeated several times, until Victor is convinced.

Of course, if Peggy knows s, the procedure proceeds without problems.

But what if Peggy doesn't know a square root of y? She can still send Victor two numbers x1 and x2 with $x1x2 \equiv y$. If she knows a square root of x1 and a square root of x2, then she knows a square root of $y \equiv x1x2$. Therefore, for at least one of them, she does not know a square root. At least half the time, Victor is going to ask her for a square root she doesn't know. Since computing square roots is hard, she is not able to produce the desired answer, and therefore Victor finds out that she doesn't know s.

Suppose, however, that Peggy predicts correctly that Victor will ask for a square root of x2. Then she chooses a random r2, computes $x2 \equiv r22 \pmod{n}$,

and lets $x_1 \equiv y x_2^{-1} \pmod{n}$. She sends x_1 and x_2 to Victor, and everything works. This method gives Peggy a 50% chance of fooling Victor on any given round, but it requires her to guess which number Victor will request each time. As soon as she fails, Victor will find out that she doesn't know s.

If Victor verifies that Peggy knows a square root, does he obtain any information that can be used by someone else? No, since in any step he is only learning the square root of a random square, not a square root of y. Of course, if Peggy uses the same random numbers more than once, he could find out the square roots of both x_1 and x_2 and hence a square root of y.

So Peggy should be careful in her choice of random numbers.

Suppose Eve is listening. She also will only learn square roots of random numbers. If she tries to use the same sequence of random numbers to masquerade as Peggy, she needs to be asked for the square roots of exactly the same sequence of x1's and x2's. If Victor asks for a square root of an x1 in place of an x2 at one step, for example, Eve will not be able to supply it.

19.2 The Feige-Fiat-Shamir Identification Scheme

The preceding protocol requires several communications between Peggy and Victor. The Feige-Fiat-Shamir method reduces this number and uses a type of parallel verification. This then is used as the basis of an identification scheme.

Again, let $n=pq$ be the product of two large primes. Peggy has secret numbers $s_1, \ldots, s_k$. Let $v_i \equiv s_i^{-2} \pmod{n}$ (we assume $\gcd(s_i, n)=1$). The numbers v_i are sent to Victor. Victor will try to verify that Peggy knows the numbers $s_1, \ldots, s_k$. Peggy and Victor proceed as follows:

1. Peggy chooses a random integer r, computes $x \equiv r^2 \pmod{n}$ and sends x to Victor.

2. Victor chooses numbers $b_1, \ldots, b_k$ with each $b_i \in \{0, 1\}$. He sends these to Peggy.

3. Peggy computes $y \equiv rs^{b_1}_1 s^{b_2}_2 \cdots s^{b_k}_k \pmod{n}$ and sends y to Victor.

4. Victor checks that $x \equiv y^2 v^{b_1}_1 v^{b_2}_2 \cdots v^{b_k}_k \pmod{n}$.

5. Steps 1 through 4 are repeated several times (each time with a different r).

Consider the case k=1. Then Peggy is asked for either r or rs_1. These are two random numbers whose quotient is a square root of v_1. Therefore, this is essentially the same idea as the simplified scheme discussed previously, with quotients instead of products.

Now let's analyze the case of larger k. Suppose, for example, that Victor sends

$b_1=1$, $b_2=1$, $b_4=1$, and all other $b_i=0$. Then Peggy must produce $y \equiv r s_1 s_2 s_4$, which is a square root of $x v_1^{-1} v_2^{-1} v_4^{-1}$. In fact, in each round, Victor is asking for a square root of a number of the form $x v_{i_1}^{-1} v_{i_2}^{-1} \cdots v_{i_j}^{-1}$. Peggy can supply a square root if she knows $r, s_{i_1}, \ldots, s_{i_j}$. If she doesn't, she will have a hard time computing a square root.

If Peggy doesn't know any of the numbers $s_1, \ldots, s_k$ (the likely scenario also if someone other than Peggy is pretending to be Peggy), she could guess the string of bits that Victor will send. Suppose she guesses correctly, before she sends x. She lets y be a random number and declares

$x \equiv y^2 v_1^{b_1} v_2^{b_2} \cdots v_k^{b_k} \pmod{n}$. When Victor sends the string of bits, Peggy sends back the value of y. Of course, the verification congruence is satisfied. But if Peggy guesses incorrectly, she will need to modify her choice of y, which means she will need some square roots of vi's.

For example, suppose Peggy is able to supply the correct response when b1=1, b2=1, b4=1, and all other bi=0. This could be accomplished by guessing the bits and using the preceding method of choosing x. However, suppose Victor sends b1=1, b3=1, and all other bi=0. Then Peggy will be ready to supply a square root of $xv_1^{-1}v_2^{-1}v_4^{-1}$ but will be asked to supply a square root of

xv1−1v3−1. This, combined with what she knows, is equivalent to knowing a square root of v2−1v3v4−1, which she is not able to compute. In an extreme case, Victor could send all bits equal to 0, which means Peggy must supply a square root of x. With Peggy's guess as before, this means she would know a square root of v1v2v4. In summary, if Peggy's guess is not correct, she will need to know the square root of a nonempty product of vi's, which she cannot compute. Therefore, there are 2k possible strings of bits that Victor can send, and only one will allow Peggy to fool Victor. In one iteration of the protocol, the chances are only one in 2k that Victor will be fooled. If the procedure is repeated t times, the

chances are 1 in 2kt that Victor is fooled. Recommended values are k=5 and t=4. Note that this gives the same probability as 20 iterations of the earlier scheme, so the present procedure is more efficient in terms of communication between Peggy and Victor. Of course, Victor has not obtained as strong a verification that Peggy knows, for example, s1, but he is very certain that Eve is not masquerading as Peggy, since Eve should not know any of the si's.

There is an interesting feature of how the numbers are arranged in Steps 1 and 4. It is possible for Peggy to use a cryptographic hash function and send the hash of x after computing it in Step

1. After Victor computes $y^{2v_1}b_1^{v_2}b_2^{\cdots}v_k^{b_k} \pmod n$ in Step 4, he can compute the hash of this number and compare with the hash sent by Peggy. The hash function is assumed to be collision resistant, so Victor can be confident that the the congruence in Step 4 is satisfied. Since the output of the hash function is probably shorter than x (for example, 256 bits for the hash, compared to 2048 bits for x), this saves a few bits of transmission.

The preceding can be used to design an identification scheme. Let I be a string that includes Peggy's name, birth date, and any other information deemed appropriate. Let H be a public hash function. A trusted authority Arthur (the

bank, a passport agency, ...) chooses n=pq to be the product of two large primes. Arthur computes H(I||j) for some small values of j, where I||j means j is appended to I. Using his knowledge of p, q, he can determine which of these numbers H(I||j) have square roots mod n and calculate a square root for each such number. This yields numbers v1=H(I||j1), …, vk=H(I||jk) and square roots s1, …, sk. The numbers I, n, j1, …, jk are made public. Arthur gives the numbers s1, …, sk to Peggy, who keeps them secret. The prime numbers p, q are discarded once the square roots are calculated. Likewise, Arthur does not need to store s1, …, sk once they are given to Peggy. These two facts add to the security, since someone

who breaks into Arthur's computer cannot compromise Peggy's security. Moreover, a different n can be used for each person, so it is hard to compromise the security of more than one individual at a time.

Note that since approximately half the numbers mod p and half the numbers mod q have square roots, the Chinese remainder theorem implies that approximately 1/4 of the numbers mod n have square roots. Therefore, each H(I||j) has a 1/4 probability of having a square root mod n. This means that Arthur should be able to produce the necessary numbers j1, …, jk quickly.

Peggy goes to an automatic teller machine, for example. The machine

reads I from Peggy's card. It downloads n, j1, …, jk from a database and calculates vi=H(I||ji) for $1 \le i \le k$. It then performs the preceding procedure to verify that Peggy knows s1, …, sk. After a few iterations, the machine is convinced that the person is Peggy and allows her to withdraw cash. A naive implementation would require a lot of typing on Peggy's part, but at least Eve won't get Peggy's secret numbers. A better implementation would use chips embedded in the card and store some information in such a way that it cannot be extracted.

If Eve obtains the communications used in the transaction, she cannot determine Peggy's secret numbers. In fact, because

of the zero-knowledge nature of the protocol, Eve obtains no information on the secret numbers s1, …, sk that can be reused in future transactions.

19.3 Exercises

1. Consider the diagram of tunnels in Figure 19.2. Suppose each of the four doors to the central chamber is locked so that a key is needed to enter, but no key is needed to exit. Peggy claims she has the key to one of the doors. Devise a zero-knowledge protocol in which Peggy proves to Victor that she can enter the central chamber. Victor should obtain no knowledge of which door Peggy can unlock.

Figure 19.2 Diagram for Exercise 1

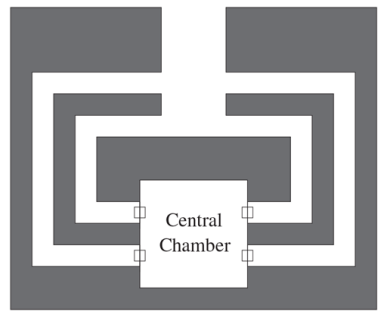


Figure 19.2 Full Alternative Text

2. Suppose p is a large prime, α is a primitive root,

and $\beta \equiv \alpha^a \pmod{p}$. The numbers p, α, β are public. Peggy wants to prove to Victor that she knows a without revealing it. They do the following:

1. Peggy chooses a random number $r \pmod{p-1}$.

2. Peggy computes $h1 \equiv \alpha^r \pmod{p}$ and $h2 \equiv \alpha^{a-r} \pmod{p}$ and sends $h1$, $h2$ to Victor.

3. Victor chooses $i=1$ or $i=2$ and asks Peggy to send either $r1=r$ or $r2=a-r \pmod{p-1}$.

4. Victor checks that $h1h2 \equiv \beta \pmod{p}$ and that $hi \equiv \alpha^{ri} \pmod{p}$.

They repeat this procedure t times, for some specified t.

1. Suppose Peggy does not know a. Why will she usually be unable to produce numbers that convince Victor?

2. If Peggy does not know a, what is the probability that Peggy can convince Victor that she knows a?

3. Suppose naive Nelson tries a variant. He

wants to convince Victor that he knows a, so he chooses a random r as before, but does not send h1, h2. Victor asks for ri and Nelson sends it. They do this several times. Why is Victor not convinced of anything? What is the essential difference between Nelson's scheme and Peggy's scheme that causes this?

3. Naive Nelson thinks he understands zero-knowledge protocols. He wants to prove to Victor that he knows the factorization of n (which equals pq for two large primes p and q) without revealing this factorization to Victor or anyone else. Nelson devises the following procedure: Victor chooses a random integer x mod n, computes $y \equiv x2 \pmod{n}$, and sends y to Nelson. Nelson computes a square root s of y mod n and sends s to Victor. Victor checks that $s2 \equiv y \pmod{n}$. Victor repeats this 20 times.

    1. Describe how Nelson computes s. You may assume that p and q are $\equiv 3 \pmod{4}$ (see Section 3.9).

    2. Explain how Victor can use this procedure to have a high probability of finding the factorization of n. (Therefore,

this is not a zero-knowledge protocol.)

3. Suppose Eve is eavesdropping and hears the values of each y and s. Is it likely that Eve obtains any useful information? (Assume no value of y repeats.)

4. Exercise 2 gave a zero-knowledge proof that Peggy knows a discrete logarithm. Here is another method. Suppose p is a large prime, α is a primitive root, and $\beta \equiv \alpha^a \pmod{p}$. The numbers p, α, β are public. Peggy wants to prove to Victor that she knows a without revealing it. They do the following:

    1. Peggy chooses a random integer k with $1 \le k < p-1$, computes $\gamma \equiv \alpha^k \pmod{p}$, and sends γ to Victor.

    2. Victor chooses a random integer r with $1 \le r < p-1$ and sends r to Peggy.

    3. Peggy computes $y \equiv k - ar \pmod{p-1}$ and sends y to Victor.

    4. Victor checks whether $\gamma \equiv \alpha^y \beta^r \pmod{p}$. If so, he believes that Peggy knows a.

    1. Show that the verification equation holds

if the procedure is followed correctly.

2. Does Victor obtain any information that will allow him to compute a?

3. Suppose Eve finds out the values of γ, r, and y. Will she be able to determine a?

4. Suppose Peggy repeats the procedure with the same value of k, but Victor uses different values r1 and r2. How can Eve, who has listened to all communications between Victor and Peggy, determine a?

The preceding procedure is the basis for the **Schnorr identification scheme**. Victor could be a bank and a could be Peggy's personal identification number. The bank stores β, and Peggy must prove she knows a to access her account. Alternatively, Victor could be a central computer and Peggy could be logging on to the computer through nonsecure telephone lines. Peggy's password is a, and the central computer stores β.

In the Schnorr scheme, p is usually chosen so that $p-1$ has a large prime factor q, and α, instead of being a primitive root, is taken to satisfy $\alpha^q \equiv 1 \pmod{p}$. The congruence defining y is then taken mod q. Moreover, r is taken to satisfy $1 \leq r \leq 2t$ for

some t, for example, t=40.

5. Peggy claims that she knows an RSA plaintext. That is, n, e, c are public and Peggy claims that she knows m such that $m^e \equiv c \pmod{n}$. She wants to prove this to Victor using a zero-knowledge protocol. Peggy and Victor perform the following steps:

    1. Peggy chooses a random integer r1 and computes $r2 \equiv m \cdot r1^{-1} \pmod{n}$ (assume that gcd(r1, n)=1.)

    2. Peggy computes $x1 \equiv r1^e \pmod{n}$ and $x2 \equiv r2^e \pmod{n}$ and sends x1, x2 to Victor.

    3. Victor checks that $x1x2 \equiv c \pmod{n}$.

    Give the remaining steps of the protocol. Victor should be at least 99% convinced that Peggy is not lying.

6. 1. Suppose that p is a large prime, and g, $h \equiv 0 \pmod{p}$. Peggy wants to prove to Victor, using a zero-knowledge protocol, that she knows a value of x with $g^x \equiv h \pmod{p}$. Peggy and Victor do the following:

1. Peggy chooses three random integers r1, r2, r3 with $r1+r2+r3 \equiv x \pmod{p-1}$.

2. Peggy computes $hi \equiv g^{ri}$, for i=1, 2, 3 and sends h1, h2, h3 to Victor.

3. Victor checks that $h1h2h3 \equiv h \pmod{n}$.

Design the remaining steps of this protocol so that Victor is at least 99% convinced that Peggy is not lying. (Note: There are two ways for Victor to proceed in Step 4. One has a higher probability of catching Peggy, if she is cheating, than the other.)

2. Give a reasonable method for Peggy to choose the three random numbers such that $r1+r2+r3= \equiv x \pmod{p-1}$. (A method that doesn't work is "Choose three random numbers and see if their sum is x. If not, try again.)

7. Suppose that n is the product of two large primes, and that s is given. Peggy wants to prove to Victor, using a zero-knowledge protocol, that she knows a

value of x with $x2 \equiv s \pmod{n}$. Peggy and Victor do the following:

1. Peggy chooses three random integers r1, r2, r3 with $r1r2r3 \equiv x \pmod{n}$.

2. Peggy computes $xi \equiv ri2$, for i=1, 2, 3 and sends x1, x2, x3 to Victor.

3. Victor checks that $x1x2x3 \equiv s \pmod{n}$.

1. Design the remaining steps of this protocol so that Victor is at least 99% convinced that Peggy is not lying. (*Note:* There are two ways for Victor to proceed in Step 4. One has a higher probability of catching Peggy, if she is cheating, than the other.)

2. Give a reasonable method for Peggy to choose the three random numbers such that $r1r2r3= \equiv x \pmod{n}$. (A method that doesn't work is "Choose three random numbers and see if their product is x. If not, try again.")

8. Peggy claims that she knows an RSA plaintext. That is, n, e, c are public and Peggy claims that she knows m such that $me \equiv c \pmod{n}$. Devise a zero-

knowledge protocol similar to that used in Exercises 6 and 7 for Peggy to convince Victor that she knows m.

Chapter 20 Information Theory

In this chapter we introduce the theoretical concepts behind the security of a cryptosystem. The basic question is the following: If Eve observes a piece of ciphertext, does she gain any new information about the encryption key that she did not already have? To address this issue, we need a mathematical definition of information. This involves probability and the use of a very important measure called entropy.

Many of the ideas in this chapter originated with Claude Shannon in the

1940s.

Before we start, let's consider an example. Roll a standard six-sided die. Let A be the event that the number of dots is odd, and let B be the event that the number of dots is at least 3. If someone tells you that the roll belongs to the event $A \cap B$, then you know that there are only two possibilities for what the roll is. In this sense, $A \cap B$ tells you more about the value of the roll than just the event A, or just the event B. In this sense, the information contained in the event $A \cap B$ is larger than the information just in A or just in B.

The idea of information is closely linked with the idea of uncertainty. Going back to the example of the die, if you are told

that the event $A \cap B$ happened, you become less uncertain about what the value of the roll was than if you are simply told that event A occurred. Thus the information increased while the uncertainty decreased. Entropy provides a measure of the increase in information or the decrease in uncertainty provided by the outcome of an experiment.

20.1 Probability Review

In this section we briefly introduce the concepts from probability needed for what follows. An understanding of probability and the various identities that arise is essential for the development of entropy.

Consider an experiment X with possible outcomes in a finite set X. For example, X could be flipping a coin and X={heads, tails}. We assume each outcome is assigned a probability. In the present example, p(X=heads)=1/2 and p(X=tails)=1/2. Often, the outcome X of an experiment is called a **random variable**.

In general, for each $x \in X$, denote the probability that $X=x$ by

$$pX(x)=px=p(X=x).$$

Note that $\sum x \in Xpx=1$. If $A \subseteq X$, let

$$p(A)=\sum x \in Apx,$$

which is the probability that X takes a value in A.

Often one performs an experiment where one is measuring several different events. These events may or may not be related, but they may be lumped together to form a new random event. For example, if we have two random events X and Y with possible outcomes X and Y, respectively, then we may create a new random event Z=(X, Y) that groups the two events together. In

this case, the new event Z has a set of possible outcomes $Z=X\times Y$, and Z is sometimes called a joint random variable.

Example

Draw a card from a standard deck. Let X be the suit of the card, so X={clubs, diamonds, hearts, spades}. Let Y be the value of the card, so Y={two, three, … , ace}. Then Z gives the 52 possibilities for the card. Note that if $x \in X$ and $y \in Y$, then $p((X, Y)=(x, y))=p(X=x, Y=y)$ is simply the probability that the card drawn has suit x and value y. Since all cards are equally probable, this probability is 1/52, which is the probability that X=x (namely 1/4) times the probability that

Y=y (namely 1/13). As we discuss later, this means X and Y are independent.

Example

Roll a die. Suppose we are interested in two things: whether the number of dots is odd and whether the number is at least 2. Let X=0 if the number of dots is even and X=1 if the number of dots is odd. Let Y=0 if the number of dots is less than 2 and Y=1 if the number of dots is at least 2. Then Z=(X, Y) gives us the results of both experiments together. Note that the probability that the number of dots is odd and less than 2 is p(Z=(1, 0))=1/6. This is not equal to p(X=0)·p(Y=0), which is (1/2)(1/6)=1/12. This means that X and Y are not independent. As we'll see, this is

closely related to the fact that knowing X gives us information about Y.

We denote

$$pX, Y(x, y)=p(X=x, Y=y).$$

Note that we can recover the probability that X=x as

$$pX(x)=\sum y \in YpX, Y(x, y).$$

We say that two random events X and Y are **independent** if

$$pX, Y(x, y)=pX(x)pY(y)$$

for all $x \in X$ and all $y \in Y$. In the preceding example, the suit of a card and the value of the card were independent.

We are also interested in the

probabilities for Y given that X=x has occurred. If pX(x)>0, define the **conditional probability** of Y=y given that X=x to be

pY(y|x)=pX, Y(x, y)pX(x).

One way to think of this is that we have restricted to the set where X=x. This has total probability pX(x)= ∑ ypX, Y(x, y). The fraction of this sum that comes from Y=y is pY(y|x).

Note that X and Y are independent if and only if

pY(y|x)=pY(y)

for all x, y. In other words, the probability of y is unaffected by what happens with X.

There is a nice way to go from the conditional probability of Y given X to the conditional probability of X given Y.

Bayes's Theorem

If $p_X(x)>0$ and $p_Y(y)>0$, then

$$p_X(x|y)=p_X(x)p_Y(y|x)p_Y(y).$$

The proof consists of simply writing the conditional probabilities in terms of their definitions.

20.2 Entropy

Roll a six-sided die and a ten-sided die. Which experiment has more uncertainty? If you make a guess at the outcome of each roll, you are more likely to be wrong with the ten-sided die than with the six-sided die. Therefore, the ten-sided die has more uncertainty. Similarly, compare a fair coin toss in which heads and tails are equally likely with a coin toss in which heads occur 90% of the time. Which has more uncertainty? The fair coin toss does, again because there is more randomness in its possibilities.

In our definition of uncertainty, we want

to make sure that two random variables X and Y that have same probability distribution have the same uncertainty. In order to do this, the measure of uncertainty must be a function only of the probability distributions and not of the names chosen for the outcomes.

We require the measure of uncertainty to satisfy the following properties:

1. To each set of nonnegative numbers $p_1, \dots, p_n$ with $p_1+\cdots+p_n=1$, the uncertainty is given by a number $H(p_1, \dots, p_n)$.

2. H should be a continuous function of the probability distribution, so a small change in the probability distribution should not drastically change the uncertainty.

3. $H(1_n, \dots, 1_n) \leq H(1_{n+1}, \dots, 1_{n+1})$ for all $n>0$. In other words, in situations where all outcomes are equally likely, the uncertainty increases when there are more possible outcomes.

4. If $0<q<1$, then

   $$H(p_1, \ldots, qp_j, (1-q)p_j, \ldots, p_n) = H(p_1, \ldots, p_j, \ldots, p_n) + p_j H(q, 1-q).$$

   What this means is that if the jth outcome is broken into two suboutcomes, with probabilities qp_j and $(1-q)p_j$, then the total uncertainty is increased by the uncertainty caused by the choice between the two suboutcomes, multiplied by the probability p_j that we are in this case to begin with. For example, if we roll a six-sided die, we can record two outcomes: *even* and *odd*. This has uncertainty $H(12, 12)$. Now suppose we break the outcome *even* into the suboutcomes 2 and {4, 6}. Then we have three possible outcomes: 2, {4, 6}, and *odd*. We have

   $$H(16, 13, 12) = H(12, 12) + 12H(23, 13).$$

   The first term is the uncertainty caused by *even* versus *odd*. The second term is the uncertainty added by splitting *even* into two suboutcomes.

Starting from these basic assumptions, Shannon [Shannon2] showed the following:

Theorem

Let H(X) be a function satisfying properties (1)–(4). In other words, for each random variable X with outcomes X={x1, …, xn} having probabilities p1, …, pn, the function H assigns a number H(X) subject to the conditions (1)–(4). Then H must be of the form

$$H(p1, \cdots, pn) = -\lambda \sum kpk \log 2pk$$

where λ is a nonnegative constant and where the sum is taken over those k such that pk>0.

Because of the theorem, we define the **entropy** of the variable X to be

$$H(X) = -\sum x \in Xp(x)\log 2p(x).$$

The entropy H(X) is a measure of the

uncertainty in the outcome of X. Note that since $\log_2 p(x) \leq 0$, we have $H(X) \geq 0$, so there is no such thing as negative uncertainty.

The observant reader might notice that there are problems when we have elements $x \in X$ that have probability 0. In this case we define $0 \log_2 0 = 0$, which is justified by looking at the limit of $x \log_2 x$ as $x \to 0$. It is typical convention that the logarithm is taken base 2, in which case entropy is measured in bits. The entropy of X may also be interpreted as the expected value of $-\log_2 p(X)$ (recall that $E[g(X)] = \sum x g(x) p(x)$).

We now look at some examples.

Example

Consider a fair coin toss. There are two outcomes, each with probability 1/2. The entropy of this random event is

$$-(12\log_2 12+12\log_2 12)=1.$$

This means that the result of the coin flip gives us 1 bit of information, or that the uncertainty in the outcome of the coin flip is 1 bit.

Example

Consider a nonfair coin toss X with probability p of getting heads and probability 1−p of getting tails (where $0<p<1$). The entropy of this event is

$$H(X)=-p\log_2 p-(1-p)\log_2(1-p).$$

If one considers H(X) as a function of p,

one sees that the entropy is a maximum when p=12. (For a more general statement, see Exercise 14.)

Example

Consider an n-sided fair die. There are n outcomes, each with probability 1/n. The entropy is

−1nlog2(1/n)−…−1nlog2(1/n)=log2(n).

There is a relationship between entropy and the number of yes-no questions needed to determine accurately the outcome of a random event. If one considers a totally nonfair coin toss where p(1)=1, then H(X)=0. This result can be interpreted as not requiring any questions to determine what the value of the event was. If someone rolls a four-

sided die, then it takes two yes-no questions to find out the outcome. For example, is the number less than 3? Is the number odd?

A slightly more subtle example is obtained by flipping two coins. Let X be the number of heads, so the possible outcomes are {0, 1, 2}. The probabilities are 1/4, 1/2, 1/4 and the entropy is

$$-\frac{1}{4}\log_2(1/4)-\frac{1}{2}\log_2(1/2)-\frac{1}{4}\log_2(1/4)=\frac{3}{2}.$$

Note that we can average 3/2 questions to determine the outcome. For example, the first question could be “Is there exactly one head?” Half of the time, this will suffice to determine the outcome. The other half of the time a second question is needed, for example, “Are there two heads?” So the average

number of questions equals the entropy.

Another way of looking at H(X) is that it measures the number of bits of information that we obtain when we are given the outcome of X. For example, suppose the outcome of X is a random 4-bit number, where each possibility has probability 1/16. As computed previously, the entropy is H(X)=4, which says we have received four bits of information when we are told the value of X.

In a similar vein, entropy relates to the minimal amount of bits necessary to represent an event on a computer (which is a binary device). See Section 20.3. There is no sense recording events whose outcomes can be predicted with

100% certainty; it would be a waste of space. In storing information, one wants to code just the uncertain parts because that is where the real information is.

If we have two random variables X and Y, the joint entropy H(X, Y) is defined as

$$H(X, Y) = -\sum_{x \in X} \sum_{y \in Y} p_{X, Y}(x, y) \log_2 p_{X, Y}(x, y).$$

This is just the entropy of the joint random variable Z=(X, Y) discussed in Section 20.1.

In a cryptosystem, we might want to know the uncertainty in a key, given knowledge of the ciphertext. This leads us to the concept of **conditional entropy**, which is the amount of uncertainty in Y, given X. It is defined

to be

$$H(Y|X)=\sum_x p_X(x)H(Y|X=x)=-\sum_x p_X(x)\left(\sum_y p_Y(y|x)\log_2 p_Y(y|x)\right)=-\sum_x\sum_y p_{X,Y}(x,y)\log_2 p_Y(y|x).$$

The last equality follows from the relationship $p_{X,Y}(x,y)=p_Y(y|x)p_X(x)$. The quantity $H(Y|X=x)$ is the uncertainty in Y given the information that X=x. It is defined in terms of conditional probabilities by the expression in parentheses on the second line. We calculate $H(Y|X)$ by forming a weighted sum of these uncertainties to get the total uncertainty in Y given that we know the value of X.

Remark

The preceding definition of conditional entropy uses the weighted average, over

the various $x \in X$, of the entropy of Y given X=x. Note that $H(Y|X) \neq -\sum x, ypY(y|x)\log2(pY(y|x))$. This sum does not have properties that information or uncertainty should have. For example, if X and Y are independent, then this definition would imply that the uncertainty of Y given X is greater than the uncertainty of Y (see Exercise 15). This clearly should not be the case.

We now derive an important tool, the chain rule for entropies. It will be useful in Section 20.4.

Theorem

(Chain Rule). $H(X, Y)=H(X)+H(Y|X)$.

Proof

$$H(X, Y) = -\sum_{x \in X} \sum_{y \in Y} p_{X,Y}(x, y) \log_2 p_{X,Y}(x, y) = -\sum_{x \in X} \sum_{y \in Y} p_{X,Y}(x, y) \log_2 p_X(x) p_Y(y|x) = -\sum_{x \in X} \sum_{y \in Y} p_{X,Y}(x, y) \log_2 p_X(x) - \sum_{x \in X} \sum_{y \in Y} p_{X,Y}(x, y) \log_2 p_Y(y|x) = \left(\sum_x \log_2 p_X(x) \sum_Y p_{X,Y}(x, y)\right) + H(Y|X) = \sum_x p_X(x) \log_2 p_X(x) + H(Y|X) \quad \text{(since } \sum_y p_{X,Y}(x, y) = p_X(x)\text{)} = H(X) + H(Y|X).$$

What does the chain rule tell us? It says that the uncertainty of the joint event (X, Y) is equal to the uncertainty of event X + uncertainty of event Y given that event X has happened.

We now state three more results about entropy.

Theorem

1. $H(X) \leq \log_2 X$, where X denotes the number of elements in X. We have equality if and only if all elements of X are equally likely.

2. $H(X, Y) \leq H(X) + H(Y)$.

3. (Conditioning reduces entropy) $H(Y|X) \leq H(Y)$, with equality if and only if X and Y are independent.

The first result states that you are most uncertain when the probability distribution is uniform. Referring back to the example of the nonfair coin flip, the entropy was maximum for p=12. This extends to events with more possible outcomes. For a proof of (1), see [Welsh, p. 5].

The second result says that the information contained in the pair (X, Y) is at most the information contained in X plus the information contained in Y. The reason for the inequality is that possibly the information supplied by X and Y overlap (which is when X and Y are not independent). For a proof of (2),

see [Stinson].

The third result is one of the most important results in information theory. Its interpretation is very simple. It says that the uncertainty one has in a random event Y given that event X occurred is less than the uncertainty in event Y alone. That is, X can only tell you information about event Y; it can't make you any more uncertain about Y.

The third result is an easy corollary of the second plus the chain rule:

$$H(X)+H(Y|X)=H(X, Y) \leq H(X)+H(Y).$$

20.3 Huffman Codes

Information theory originated in the late 1940s from the seminal papers by Claude Shannon. One of the primary motivations behind Shannon's mathematical theory of information was the problem of finding a more compact way of representing data. In short, he was concerned with the problem of compression. In this section we briefly touch on the relationship between entropy and compression and introduce Huffman codes as a method for more succinctly representing data.

For more on how to compress data, see [Cover-Thomas] or [Nelson-Gailly].

Example

Suppose we have an alphabet with four letters a, b, c, d, and suppose these letters appear in a text with frequencies as follows.

a	b	c	d
.5	.3	.1	.1

We could represent a as the binary string 00, b as 01, c as 10, and d as 11. This means that the message would average two bits per letter. However, suppose we represent a as 1, b as 01, c as 001, and d as 000. Then the average number of bits per letter is

(1)(.5)+(2)(.3)+(3)(.1)+(3)(.1)=1.7

(the number of bits for a times the

frequency of a, plus the number of bits for b times the frequency of b, etc.). This encoding of the letters is therefore more efficient.

In general, we have a random variable with outputs in a set X. We want to represent the outputs in binary in an efficient way; namely, the average number of bits per output should be as small as possible.

An early example of such a procedure is Morse code, which represents letters as sequences of dots and dashes and was developed to send messages by telegraph. Morse asked printers which letters were used most, and made the more frequent letters have smaller representations. For example, e is

represented as · and t as –. But x is –··– and z is ––··.

A more recent method was developed by Huffman. The idea is to list all the outputs and their probabilities. The smallest two are assigned 1 and 0 and then combined to form an output with a larger probability. The same procedure is then applied to the new list, assigning 1 and 0 to the two smallest, then combining them to form a new list. This procedure is continued until there is only one output remaining. The binary strings are then obtained by reading backward through the procedure, recording the bits that have been assigned to a given output and to combinations containing it. This is best

explained by an example.

Suppose we have outputs a, b, c, d with probabilities 0.5, 0.3, 0.1, 0.1, as in the preceding example. The diagram in Figure 20.1 gives the procedure.

Figure 20.1 An Example of Huffman Encoding

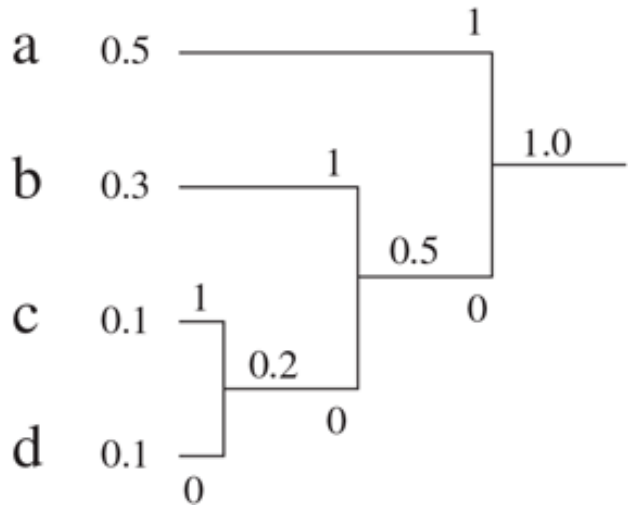


Figure 20.1 Full Alternative Text

Note that when there were two choices for the lowest, we made a random

choice for which one received 0 and which one received 1. Tracing backward through the table, we see that a only received a 1, b received 01, c received 001, and d received 000. These are exactly the assignments made previously that gave a low number of bits per letter.

A useful feature of Huffman encoding is that it is possible to read a message one letter at a time. For example, the string 011000 can only be read as bad; moreover, as soon as we have read the first two bits 01, we know that the first letter is b.

Suppose instead that we wrote the bits assigned to letters in reverse order, so b is 10 and c is 001. Then the message

101000 cannot be determined until all bits have been read, since it potentially could start with bb or ba.

Even worse, suppose we had assigned 0 to a instead of 1. Then the messages aaa and d would be the same. It is possible to show that Huffman encoding avoids these two problems.

The average number of bits per output is closely related to the entropy.

Theorem

Let L be the average number of bits per output for Huffman encoding for the random variable X. Then

$H(X) \leq L < H(X)+1.$

This result agrees with the interpretation that the entropy measures how many bits of information are contained in the output of X. We omit the proof. In our example, the entropy is

$$H(X) = -(.5\log_2(.5) + .3\log_2(.3) + .1\log(.1) + .1\log(.1)) \approx 1.685.$$

20.4 Perfect Secrecy

Intuitively, the one-time pad provides perfect secrecy. In Section 4.4, we gave a mathematical meaning to this statement. In the present section, we repeat some of the arguments of that section and phrase some of the ideas in terms of entropy.

Suppose we have a cipher system with possible plaintexts P, ciphertexts C, and keys K. Each plaintext in P has a certain probability of occurring; some are more likely than others. The choice of a key in K is always assumed to be independent of the choice of plaintext. The possible ciphertexts in C have

various probabilities, depending on the probabilities for P and K.

If Eve intercepts a ciphertext, how much information does she obtain for the key? In other words, what is H(K|C)? Initially, the uncertainty in the key was H(K). Has the knowledge of the ciphertext decreased the uncertainty?

Example

Suppose we have three possible plaintexts: a, b, c with probabilities .5, .3, .2 and two keys k1, k2 with probabilities .5 and .5. Suppose the possible ciphertexts are U, V, W. Let ek be the encryption function for the key k. Suppose

ek1(a)=U, ek1(b)=V, ek1(c)=Wek2(a)=U, ek2(b)=W, ek2(c)=V.

Let pP(a) denote the probability that the plaintext is a, etc. The probability that the ciphertext is U is

pC(U)=pK(k1)pP(a)+pK(k2)pP(a)=(.5)(.5)+(.5)(.5)=.50.

Similarly, we calculate pC(V)=.25 and pC(W)=.25.

Suppose someone intercepts a ciphertext. This gives some information on the plaintext. For example, if the ciphertext is U, then it can be deduced immediately that the plaintext was a. If the ciphertext is V, the plaintext was either b or c.

We can even say more: The probability that a ciphertext is V is .25, so the conditional probability that the plaintext was b, given that the ciphertext is V is

$$p(b|V)=p(P, C)(b, V)pC(V)=p(P, K)(b, k1)pC(V)=(.3)(.5).25=.6.$$

Similarly, $p(c|V)=.4$ and $p(a|V)=0$. We can also calculate

$$p(a|W)=0,\ p(b|W)=.6,\ p(c|W)=.4.$$

Note that the original probabilities of the plaintexts were .5, .3, and .2; knowledge of the ciphertext allows us to revise the probabilities. Therefore, the ciphertext gives us information about the plaintext. We can quantify this via the concept of conditional entropy. First, the entropy of the plaintext is

$$H(P)=-(.5\log2(.5)+.3\log2(.3)+.2\log2(.2))=1.485.$$

The conditional entropy of P given C is

$$H(P|C)=-\sum x\in\{a, b, c\}\sum Y\in\{U, V, W\}p(Y)p(x|Y)\log2(p(x|Y))=.485.$$

Therefore, in the present example, the

uncertainty for the plaintext decreases when the ciphertext is known.

On the other hand, we suspect that for the one-time pad the ciphertext yields no information about the plaintext that was not known before. In other words, the uncertainty for the plaintext should equal the uncertainty for the plaintext given the ciphertext. This leads us to the following definition and theorem.

Definition

A cryptosystem has **perfect secrecy** if $H(P|C)=H(P)$.

Theorem

The one-time pad has perfect secrecy.

Proof. Recall that the basic setup is the following: There is an alphabet with Z letters (for example, Z could be 2 or 26). The possible plaintexts consist of strings of characters of length L. The ciphertexts are strings of characters of length L. There are ZL keys, each consisting of a sequence of length L denoting the various shifts to be used. The keys are chosen randomly, so each occurs with probability 1/ZL.

Let $c \in C$ be a possible ciphertext. As before, we calculate the probability that c occurs:

$$pC(c)= \sum x \in P, k \in Kek(x)=cpP(x)pK(k).$$

Here ek(x) denotes the ciphertext obtained by encrypting x using the key k. The sum is over those pairs x, k such

that k encrypts x to c. Note that we have used the independence of P and K to write joint probability p(P, K)(x, k) as the product of the individual probabilities.

In the one-time pad, every key has equal probability 1/ZL, so we can replace pK(k) in the above sum by 1/ZL. We obtain

pC(c)=1ZL ∑ x ∈ P, k ∈ Kek(x)=cpP(x).

We now use another important feature of the one-time pad: For each plaintext x and each ciphertext c, there is exactly one key k such that ek(x)=c. Therefore, every x ∈ P occurs exactly once in the preceding sum, so we have Z −L ∑ x ∈ PpP(x). But the sum of the probabilities of all possible plaintexts is

1, so we obtain

$$p_C(c) = \frac{1}{ZL}.$$

This confirms what we already suspected: Every ciphertext occurs with equal probability.

Now let's calculate some entropies. Since K and C each have equal probabilities for all ZL possibilities, we have

$$H(K) = H(C) = \log_2(ZL).$$

We now calculate H(P, K, C) in two different ways. Since knowing (P, K, C) is the same as knowing (P, K), we have

$$H(P, K, C) = H(P, K) = H(P) + H(K).$$

The last equality is because P and K are independent. Also, knowing (P, K, C) is

the same as knowing (P, C) since C and P determine K for the one-time pad. Therefore,

$$H(P, K, C)=H(P, C)=H(P|C)+H(C).$$

The last equality is the chain rule. Equating the two expressions, and using the fact that $H(K)=H(C)$, we obtain $H(P|C)=H(P)$. This proves that the one-time pad has perfect secrecy.

The preceding proof yields the following more general result. Let #K denote the number of possible keys, etc.

Theorem

Consider a cryptosystem such that

1. Every key has probability 1/#K.

2. For each $x \in P$ and $c \in C$ there is exactly one $k \in K$ such that $e_k(x)=c$.

Then this cryptosystem has perfect secrecy.

It is easy to deduce from condition (2) that #C=#K. Conversely, it can be shown that if #P=#C=#K and the system has perfect secrecy, then (1) and (2) hold (see [Stinson, Theorem 2.4]).

It is natural to ask how the preceding concepts apply to RSA. The possibly surprising answer is that $H(P|C)=0$; namely, the ciphertext determines the plaintext. The reason is that entropy does not take into account computation time. The fact that it might take billions of years to factor n is irrelevant. What

counts is that all the information needed to recover the plaintext is contained in the knowledge of n, e, and c.

The more relevant concept for RSA is the computational complexity of breaking the system.

20.5 The Entropy of English

In an English text, how much information is obtained per letter? If we had a random sequence of letters, each appearing with probability 1/26, then the entropy would be log2(26)=4.70; so each letter would contain 4.7 bits of information. If we include spaces, we get log2(27)=4.75. But the letters are not equally likely: a has frequency .082, b has frequency .015, etc. (see Section 2.3). Therefore, we consider

$$-(.082\log 2.082+.015\log 2.015+\cdots)=4.18.$$

However, this doesn't tell the whole story. Suppose we have the sequence of letters *we are studyin*. There is very

little uncertainty as to what the last letter is; it is easy to guess that it is g. Similarly, if we see the letter q, it is extremely likely that the next letter is u. Therefore, the existing letters often give information about the next letter, which means that there is not as much additional information carried by that letter. This says that the entropy calculated previously is still too high. If we use tables of the frequencies of the 262=676 digrams (a digram is a two-letter combination), we can calculate the conditional entropy of one letter, given the preceding letter, to be 3.56. Using trigram frequencies, we find that the conditional entropy of a letter, given the preceding two letters, is approximately 3.3. This means that, on the average, if

we know two consecutive letters in a text, the following letter carries 3.3 bits of additional information. Therefore, if we have a long text, we should expect to be able to compress it at least by a factor of around 3.3/4.7=.7.

Let L represent the letters of English. Let LN denote the N-gram combinations. Define the entropy of English to be

$$H_{English} = \lim_{N \to \infty} H(L_N)/N,$$

where H(LN) denotes the entropy of N-grams. This gives the average amount of information per letter in a long text, and it also represents the average amount of uncertainty in guessing the next letter, if we already know a lot of the text. If the letters were all independent of each

other, so the probability of the digram qu equaled the probability of q times the probability of u, then we would have $H(LN)=N \cdot H(L)$, and the limit would be $H(L)$, which is the entropy for one-letter frequencies. But the interactions of letters, as noticed in the frequencies for digrams and trigrams, lower the value of $H(LN)$.

How do we compute $H(LN)$? Calculating 100-gram frequencies is impossible. Even tabulating the most common of them and getting an approximation would be difficult. Shannon proposed the following idea.

Suppose we have a machine that is an optimal predictor, in the sense that, given a long string of text, it can

calculate the probabilities for the letter that will occur next. It then guesses the letter with highest probability. If correct, it notes the letter and writes down a 1. If incorrect, it guesses the second most likely letter. If correct, it writes down a 2, etc. In this way, we obtain a sequence of numbers. For example, consider the text itissunnytoday. Suppose the predictor says that t is the most likely for the 1st letter, and it is wrong; its second guess is i, which is correct, so we write the i and put 2 below it. The predictor then predicts that t is the next letter, which is correct. We put 1 beneath the t. Continuing, suppose it finds i on its 1st guess, etc. We obtain a situation like the following:

itissunnytoday21114321411111

Using the prediction machine, we can reconstruct the text. The prediction machine says that its second guess for the first letter will be i, so we know the 1st letter is i. The predictor says that its first guess for the next letter is t, so we know that's next. The first guess for the next is i, etc.

What this means is that if we have a machine for predicting, we can change a text into a string of numbers without losing any information, because we can reconstruct the text. Of course, we could attempt to write a computer program to do the predicting, but Shannon suggested that the best predictor is a person who speaks English. Of course, a person is unlikely to be as deterministic

as a machine, and repeating the experiment (assuming the person forgets the text from the first time) might not yield an identical result. So reconstructing the text might present a slight difficulty. But it is still a reasonable assumption that a person approximates an optimal predictor.

Given a sequence of integers corresponding to a text, we can count the frequency of each number. Let

q_i= frequency of the number i.

Since the text and the sequence of numbers can be reconstructed from each other, their entropies must be the same. The largest the entropy can be for the sequence of numbers is when these numbers are independent. In this case,

the entropy is $-\sum i=126qilog2(qi)$. However, the numbers are probably not independent. For example, if there are a couple consecutive 1s, then perhaps the predictor has guessed the rest of the word, which means that there will be a few more 1s. However, we get an upper bound for the entropy, which is usually better than the one we obtain using frequencies of letters. Moreover, Shannon also found a lower bound for the entropy. His results are

$$\sum 1=126i(qi-qi+1)log2(i) \leq HEnglish \leq -\sum i=126qilog2(qi).$$

Actually, these are only approximate upper and lower bounds, since there is experimental error, and we are really considering a limit as $N \to \infty$.

These results allow an experimental

estimation of the entropy of English. Alice chooses a text and Bob guesses the first letter, continuing until the correct guess is made. Alice records the number of guesses. Bob then tries to guess the second letter, and the number of guesses is again recorded. Continuing in this way, Bob tries to guess each letter. When he is correct, Alice tells him and records the number of guesses. Shannon gave Table 20.1 as a typical result of an experiment. Note that he included spaces, but ignored punctuation, so he had 27 possibilities: There are 102 symbols. There are seventy-nine 1s, eight 2s, three 3s, etc. This gives

q1=79/102, q2=8/102, q3=3/102, q4=q5=2/102, q6=3/102, q7=q8=q11=q15=q

Table 20.1 Shannon’s Experiment on the Entropy of English

t	h	e	r	e		i	s		n	o		r	e	v	e	r	s	e
1	1	1	5	1	1	2	1	1	2	1	1	15	1	17	1	1	1	2	1

o	n		a		m	o	t	o	r	c	y	c	l	e		a		
3	2	1	2	2	7	1	1	1	1	4	1	1	1	1	1	3	1	

f	r	i	e	n	d		o	f		m	i	n	e		f	o	u	n	d	
8	6	1	3	1	1	1	1	1	1	1	1	1	1	1	6	2	1	1	1	1

t	h	i	s		o	u	t		r	a	t	h	e	r				
1	1	2	1	1	1	1	1	1	4	1	1	1	1	1	1			

d	r	a	m	a	t	i	c	a	l	l	y		t	h	e			
11	5	1	1	1	1	1	1	1	1	1	1	1	6	1	1	1		

o	t	h	e	r		d	a	y										
1	1	1	1	1	1	1	1	1	1									

Table 20.1 Full Alternative Text

The upper bound for the entropy is therefore

$$-(79102\log_2 79102+\cdots+1102\log_2 1102) \approx 1.42.$$

Note that since we are using $0\log_2 0=0$, the terms with $q_i=0$ can be omitted. The lower bound is

$$1\cdot(79102-8102)\log_2(1)+2\cdot(8102-3102)\log_2(2)+\ldots \approx .72.$$

A reasonable estimate is therefore that

the entropy of English is near 1, maybe slightly more than 1.

If we want to send a long English text, we could write each letter (and the space) as a string of five bits. This would mean that a text of length 102, such as the preceding, would require 510 bits. It would be necessary to use something like this method if the letters were independent and equally likely. However, suppose we do a Huffman encoding of the message
1, 1, 1, 5, 1, 1, 2, … from Table 20.1. Let

1↔12↔1103↔10104↔01005↔111006↔00107↔011008↔1100011↔0100015↔1

All other numbers up to 27 can be represented by various combinations of six or more bits. To send the message

requires

$$79\cdot 1+8\cdot 3+3\cdot 4+2\cdot 4+\cdots+1\cdot 6=171 \text{ bits},$$

which is 1.68 bits per letter.

Note that five bits per letter is only slightly more than the "random" entropy 4.75, and 1.68 bits per letter is slightly more than our estimate of the entropy of English. These agree with the result that entropy differs from the average length of a Huffman encoding by at most 1.

One way to look at the preceding entropy calculations is to say that English is around 75% redundant. Namely, if we send a long message in standard written English, compared to the optimally compressed text, the ratio is approximately 4 to 1 (that is, the

random entropy 4.75 divided by the entropy of English, which is around 1). In our example, we were close, obtaining a ratio near 3 to 1 (namely 4.75/1.68).

Define the **redundancy** of English to be

$$R = 1 - H_{English} \log_2(26).$$

Then R is approximately 0.75, which is the 75% redundancy mentioned previously.

20.5.1 Unicity Distance

Suppose we have a ciphertext. How many keys will decrypt it to something meaningful? If the text is long enough, we suspect that there is a unique key and a unique corresponding plaintext. The

unicity distance n0 for a cryptosystem is the length of ciphertext at which one expects that there is a unique meaningful plaintext. A rough estimate for the unicity distance is

$$n0 = \log 2|K| R \log 2|L|,$$

where $|K|$ is the number of possible keys, $|L|$ is the number of letters or symbols, and R is the redundancy (see [Stinson]). We'll take R=.75 (whether we include spaces in our language or not; the difference is small).

For example, consider the substitution cipher, which has 26! keys. We have

$$n0 = \log 226!.75 \log 226 \approx 25.1.$$

This means that if a ciphertext has length 25 or more, we expect that

usually there is only one possible meaningful plaintext. Of course, if we have a ciphertext of length 25, there are probably several letters that have not appeared. Therefore, there could be several possible keys, all of which decrypt the ciphertext to the same plaintext.

As another example, consider the affine cipher. There are 312 keys, so

$$n_0 = \log_{26} 312.75 \log_{26} \approx 2.35.$$

This should be regarded as only a very rough approximation. Clearly it should take a few more letters to get a unique decryption. But the estimate of 2.35 indicates that very few letters suffice to yield a unique decryption in most cases for the affine cipher.

Finally, consider the one-time pad for a message of length N. The encryption is a separate shift mod 26 for each letter, so there are 26N keys. We obtain the estimate

$$n_0 \approx \log_{26} 26^N .75 \log_{26} = 1.33N.$$

In this case, it says we need more letters than the entire ciphertext to get a unique decryption. This reflects the fact that all plaintexts are possible for any ciphertext.

20.6 Exercises

1. Let X1 and X2 be two independent tosses of a fair coin. Find the entropy H(X1) and the joint entropy H(X1, X2). Why is H(X1, X2)=H(X1)+H(X2)?

2. Consider an unfair coin where the two outcomes, heads and tails, have probabilities p(heads)=p and p(tails)=1−p.

    1. If the coin is flipped two times, what are the possible outcomes along with their respective probabilities?

    2. Show that the entropy in part (a) is −2plog2(p)−2(1−p)log2(1−p). How could this have been predicted without calculating the probabilities in part (a)?

3. A random variable X takes the values 1, 2, ⋯, n, ⋯ with probabilities 12, 122, ⋯, 12n, ⋯. Calculate the entropy H(X).

4. Let X be a random variable taking on integer values. The probability is 1/2 that X is in the range [0, 28−1], with all such values being equally

likely, and the probability is 1/2 that the value is in the range [28, 232−1], with all such values being equally likely. Compute H(X).

5. Let X be a random event taking on the values −2, −1, 0, 1, 2, all with positive probability. What is the general inequality/equality between H(X) and H(Y), where Y is the following?

    1. Y=2X

    2. Y=X2

6.

    1. In this problem we explore the relationship between the entropy of a random variable X and the entropy of a function f(X) of the random variable. The following is a short proof that shows H(f(X)) ≤ H(X). Explain what principles are used in each of the steps.

        H(X, f(X))=H(X)+H(f(X)|X)=H(X), H(X, f(X))=H(f(X))+H(X|f(X)) ≥ H(f(X)).

    2. Letting X take on the values ±1 and letting f(x)=x2, show that it is possible to have H(f(X))<H(X).

    3. In part (a), show that you have equality if

and only if f is a one-to-one function (more precisely, f is one-to-one on the set of outputs of X that have nonzero probability).

4. The preceding results can be used to study the behavior of the run length coding of a sequence. Run length coding is a technique that is commonly used in data compression. Suppose that $X_1, X_2, \cdots, X_n$ are random variables that take the values 0 or 1. This sequence of random variables can be thought of as representing the output of a binary source. The run length coding of $X_1, X_2, \cdots, X_n$ is a sequence $L=(L_1, L_2, \cdots, L_k)$ that represents the lengths of consecutive symbols with the same value. For example, the sequence 110000100111 has a run length sequence of $L=(2, 4, 1, 2, 3)$. Observe that $\mathbf{L}$ is a function of $X_1, X_2, \cdots, X_n$. Show that $\mathbf{L}$ and X_1 uniquely determine $X_1, X_2, \cdots, X_n$. Do $\mathbf{L}$ and X_n determine $X_1, X_2, \cdots, X_n$? Using these observations and the preceding results, compare $H(X_1, X_2, \cdots, X_n)$, $H(L)$, and $H(L, X_1)$.

7. A bag contains five red balls, three white balls, and two black balls that are identical to each other in every manner except color.

    1. Choose two balls from the bag with replacement. What is the entropy of this experiment?

    2. What is the entropy of choosing two balls without replacement? (Note: In both parts, the order matters; i.e., red then white is not the same as white then red.)

8. We often run into situations where we have a sequence of n random events. For example, a piece of text is a long sequence of letters. We are concerned with the rate of growth of the joint entropy as n increases. Define the entropy rate of a sequence X={Xk} of random events as

    $H\infty(X)=\lim n\to\infty 1nH(X1, X2, \cdots Xn).$

    1. A very crude model for a language is to assume that subsequent letters in a piece of text are independent and come from identical probability distributions. Using this, show that the entropy rate equals H(X1).

2. In general, there is dependence among the random variables. Assume that X1, X2, ⋯Xn have the same probability distribution but are somehow dependent on each other (for example, if I give you the letters TH you can guess that the next letter is E). Show that

   $$H(X1, X2, \cdots Xn) \leq \sum kH(Xk)$$

   and thus that

   $$H\infty(X) \leq H(X1)$$

   (if the limit defining $H\infty$ exists).

9. Suppose we have a cryptosystem with only two possible plaintexts. The plaintext a occurs with probability 1/3 and b occurs with probability 2/3. There are two keys, k1 and k2, and each is used with probability 1/2. Key k1 encrypts a to A and b to B. Key k2 encrypts a to B and b to A.

   1. Calculate H(P), the entropy for the plaintext.

   2. Calculate H(P|C), the conditional entropy for the plaintext given the ciphertext. (*Optional hint:* This can be done with no additional calculation by matching up this system with another well-known

system.)

10. Consider a cryptosystem $\{P, K, C\}$.

    1. Explain why
       $H(P, K)=H(C, P, K)=H(P)+H(K)$.

    2. Suppose the system has perfect secrecy. Show that

       $H(C, P)=H(C)+H(P)$

       and

       $H(C)=H(K)-H(K|C, P)$.

    3. Suppose the system has perfect secrecy and, for each pair of plaintext and ciphertext, there is at most one corresponding key that does the encryption. Show that $H(C)=H(K)$.

11. Prove that for a cryptosystem $\{P, K, C\}$ we have

    $$H(C|P)=H(P, K, C)-H(P)-H(K|C, P)=H(K)-H(K|C, P).$$

12. Consider a Shamir secret sharing scheme where any five people of a set of 20 can determine the secret K, but no fewer can do so. Let $H(K)$ be the

entropy of the choice of K, and let H(K|S1) be the conditional entropy of K, given the information supplied to the first person. What are the relative sizes of H(K) and H(K|S1)? (Larger, smaller, equal?)

13. Let X be a random event taking on the values 1, 2, 3, …, 36, all with equal probability.

    1. What is the entropy H(X)?

    2. Let Y=X36(mod37). What is H(Y)?

14.

    1. Show that the maximum of $-p\log_2 p - (1-p)\log_2(1-p)$ for $0 \le p \le 1$ occurs when $p=1/2$.

    2. Let $p_i \ge 0$ for $1 \le i \le n$. Show that the maximum of

       $$-\sum_i p_i \log_2 p_i,$$

       subject to the constraint $\sum_i p_i = 1$, occurs when $p_1 = \cdots = p_n$. (Hint: Lagrange multipliers could be useful in this problem.)

15.

    1. Suppose we define $\tilde{H}(Y|X)=$

$-\sum x, ypY(y|x)log2pY(y|x)$. Show that if X and Y are independent, and X has $|X|$ possible outputs, then $H\tilde{}(Y|X)=|X|$ $H(Y) \geq H(Y)$.

2. Use (a) to show that $H\tilde{}(Y|X)$ is not a good description of the uncertainty of Y given X.

Chapter 21 Elliptic Curves

In the mid-1980s, Miller and Koblitz introduced elliptic curves into cryptography, and Lenstra showed how to use elliptic curves to factor integers. Since that time, elliptic curves have played an increasingly important role in many cryptographic situations. One of their advantages is that they seem to offer a level of security comparable to classical cryptosystems that use much larger key sizes. For example, it is estimated in [Blake et al.] that certain conventional systems with a 4096-bit key size can be replaced by 313-bit elliptic curve systems. Using much shorter numbers can represent a

considerable savings in hardware implementations.

In this chapter, we present some of the highlights. For more details on elliptic curves and their cryptologic uses, see [Blake et al.], [Hankerson et al.], or [Washington]. For a list of elliptic curves recommended by NIST for cryptographic uses, see [FIPS 186-2].

21.1 The Addition Law

An elliptic curve E is the graph of an equation

$$E: y^2 = x^3 + ax^2 + bx + c,$$

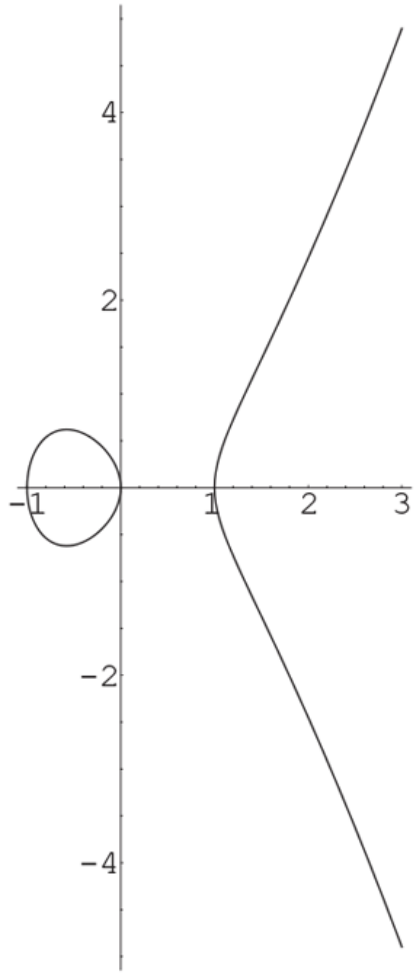
where a, b, c are in whatever is the appropriate set (rational numbers, real numbers, integers mod p, etc.). In other words, let K be the rational numbers, the real numbers, or the integers mod a prime p (or, for those who know what this means, any field of characteristic not 2; but see Section 21.4). Then we assume $a, b, c \in K$ and take E to be

$$\{(x, y) | x, y \in K, y^2 = x^3 + ax^2 + bx + c\}.$$

As will be discussed below, it is also

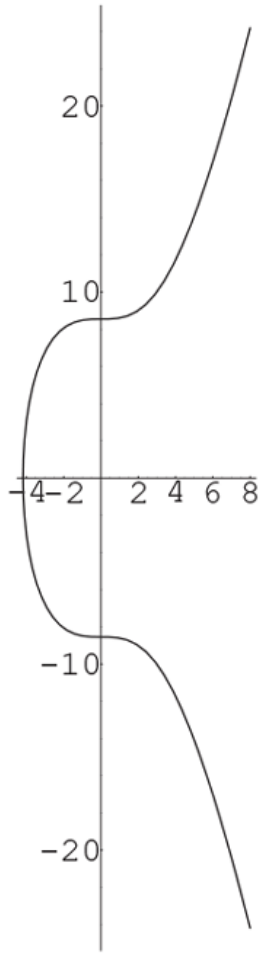
convenient to include a point (∞, ∞), which often will be denoted simply by ∞.

Let's consider the case of real numbers first, since this case allows us to work with pictures. The graph E has two possible forms, depending on whether the cubic polynomial has one real root or three real roots. For example, the graphs of $y2=x(x+1)(x-1)$ and $y2=x3+73$ are the following:



$$y^2 = x(x+1)(x-1)$$

21.1-1 Full Alternative Text



$$y^2 = x^3 + 73$$

21.1-2 Full Alternative Text

The case of two components (for example, y2=x(x+1)(x−1)) occurs when the cubic polynomial has three real

roots. The case of one component (for example, y2=x3+73) occurs when the cubic polynomial has only one real root.

For technical reasons that will become clear later, we also include a **"point at infinity,"** denoted ∞, which is most easily regarded as sitting at the top of the y-axis. It can be treated rigorously in the context of projective geometry (see [Washington]), but this intuitive notion suffices for what we need. The bottom of the y-axis is identified with the top, so ∞ also sits at the bottom of the y-axis.

Now let's look at elliptic curves mod p, where p is a prime. For example, let E be given by

$y2 \equiv x3+2x-1 (\mathrm{mod}\, 5)$.

We can list the points on E by letting x run through the values 0, 1, 2, 3, 4 and solving for y:

$(0, 2), (0, 3), (2, 1), (2, 4), (4, 1), (4, 4), \infty$.

Note that we again include a point ∞.

Elliptic curves mod p are finite sets of points. It is these elliptic curves that are useful in cryptography.

Technical point: We assume that the cubic polynomial x3+ax2+bx+c has no multiple roots. This means we exclude, for example, the graph of y2=(x−1)2(x +2). Such curves will be discussed in Subsection 21.3.1.

Technical point: For most situations,

equations of the form $y^2=x^3+bx+c$ suffice for elliptic curves. In fact, in situations where we can divide by 3, a change of variables changes an equation $y^2=x^3+ax^2+bx+c$ into an equation of the form $y^2=x^3+b'x+c'$. See Exercise 1. However, sometimes it is necessary to consider elliptic curves given by equations of the form

$$y^2+a_1xy+a_3y=x^3+a_2x^2+a_4x+a_6,$$

where $a_1, \ldots, a_6$ are constants. If we are working mod p, where $p>3$ is prime, or if we are working with real, rational, or complex numbers, then simple changes of variables transform the present equation into the form $y^2=x^3+bx+c$. However, if we are working mod 2 or mod 3, or with a finite

field of characteristic 2 or 3 (that is, 1+1=0 or 1+1+1=0), then we need to use the more general form. Elliptic curves over fields of characteristic 2 will be mentioned briefly in Section 21.4.

Historical point: Elliptic curves are not ellipses. They received their name from their relation to *elliptic integrals* such as

$$\int z1z2dxx3+bx+cand \int z1z2x\ dxx3+bx+c$$

that arise in the computation of the arc length of ellipses.

The main reason elliptic curves are important is that we can use any two points on the curve to produce a third point on the curve. Given points P1 and P2 on E, we obtain a third point P3 on E as follows (see Figure 21.1): Draw the

line L through P1 and P2 (if P1=P2, take the tangent line to E at P1). The line L intersects E in a third point Q. Reflect Q through the x-axis (i.e., change y to −y) to get P3. Define a law of addition on E by

P1+P2=P3.

Figure 21.1 Adding Points on an Elliptic Curve

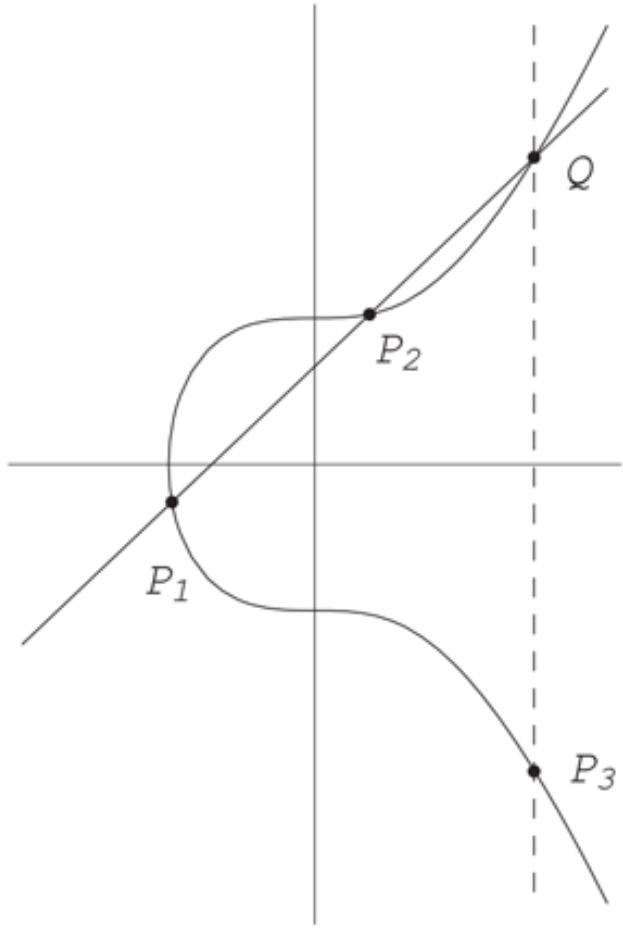


Figure 21.1 Full Alternative Text

Note that this is not the same as adding points in the plane.

Example

Suppose E is defined by y2=x3+73. Let P1=(2, 9) and P2=(3, 10). The line L through P1 and P2 is

y=x+7.

Substituting into the equation for E yields

(x+7)2=x3+73,

which yields x3−x2−14x+24=0. Since L intersects E in P1 and P2, we already know two roots, namely x=2 and x=3. Moreover, the sum of the three roots is minus the coefficient of x2 (Exercise 1) and therefore equals 1. If x is the third root, then

2+3+x=1,

so the third point of intersection has x= −4. Since y=x+7, we have y=3, and

Q=(−4, 3). Reflect across the x-axis to obtain

(2, 0)+(3, 10)=P3=(−4, −3).

Now suppose we want to add P3 to itself. The slope of the tangent line to E at P3 is obtained by implicitly differentiating the equation for E:

2ydy=3x2dx, sodydx=3x22y=−8,

where we have substituted (x, y)=(−4, −3) from P3. In this case, the line L is y=−8(x+4)−3. Substituting into the equation for E yields

(−8(x+4)−3)2=x3+73,

hence x3−(−8)2x2+⋯=0. The sum of the three roots is 64 (= minus the coefficient of x2). Because the line L is tangent to E, it follows that x=−4 is a

double root. Therefore,

$$(-4)+(-4)+x=64,$$

so the third root is x=72. The corresponding value of y (use the equation of L) is −611. Changing y to −y yields

$$P3+P3=(72, 611).$$

What happens if we try to compute P + ∞ ? We make the convention that the lines through ∞ are vertical. Therefore, the line through P=(x, y) and ∞ intersects E in P and also in (x, −y). When we reflect (x, −y) across the x-axis, we get back P=(x, y). Therefore,

$$P+\infty=P.$$

We can also subtract points. First, observe that the line through (x, y) and

$(x, -y)$ is vertical, so the third point of intersection with E is ∞. The reflection across the x-axis is still ∞ (that's what we meant when we said ∞ sits at the top and at the bottom of the y-axis). Therefore,

$$(x, y)+(x, -y)=\infty.$$

Since ∞ plays the role of an additive identity (in the same way that 0 is the identity for addition with integers), we define

$$-(x, y)=(x, -y).$$

To subtract points $P-Q$, simply add P and $-Q$.

Another way to express the addition law is to say that

P+Q+R=∞⇔P, Q, R are collinear.

(See Exercise 17.)

For computations, we can ignore the geometrical interpretation and work only with formulas, which are as follows:

Addition Law

Let E be given by y2=x3+bx+c and let

P1=(x1, y1), P2=(x2, y2).

Then

P1+P2=P3=(x3, y3),

where

x3=m2−x1−x2y3=m(x1−x3)−y1

and

m=(y2−y1)/(x2−x1)ifP1 ≠ P2(3x12+b)/(2y1)ifP1=P2.

If the slope m is infinite, then P3= ∞ . There is one additional law: ∞ +P=P for all points P.

It can be shown that the addition law is associative:

(P+Q)+R=P+(Q+R).

It is also commutative:

P+Q=Q+P.

When adding several points, it therefore doesn’t matter in what order the points are added nor how they are grouped together. In technical terms, we have found that the points of E form an abelian group. The point ∞ is the identity element of this group.

If k is a positive integer and P is a point on an elliptic curve, we can define

kP=P+P+⋯+P(k summands).

We can extend this to negative k. For example, (−3)P=3(−P)=(−P)+(−P)+(−P), where −P is the reflection of P across the x-axis. The associative law means that we can group the summands in any way we choose when computing a multiple of a point. For example, suppose we want to compute 100P. We do the additive version of successive squaring that was used in modular exponentiation:

2p=P+P4P=2P+2P8P=4P+4P16P=8P+8P32P=16P+16P64P=32P+32P100P=64P+32P+4P.

The associative law means, for example, that 4P can be computed as 2P+2P=(P

+P)+(P+P). It also could have been computed in what might seem to be a more natural way as ((P+P)+P)+P, but this is slower because it requires three additions instead of two.

For more examples, see Examples 41–44 in the Computer Appendices.

21.2 Elliptic Curves Mod *p*

If p is a prime, we can work with elliptic curves mod p using the aforementioned ideas. For example, consider

$$E : y^2 \equiv x^3+4x+4 \pmod 5.$$

The points on E are the pairs (x, y) mod 5 that satisfy the equation, along with the point at infinity. These can be listed as follows. The possibilities for x mod 5 are 0, 1, 2, 3, 4. Substitute each of these into the equation and find the values of y that solve the equation:

$$x \equiv 0 \Longrightarrow y^2 \equiv 4 \Longrightarrow y \equiv 2, 3 \pmod 5 \quad x \equiv 1 \Longrightarrow y^2 \equiv 9 \equiv 4 \Longrightarrow y \equiv 2, 3 \pmod 5 \quad x \equiv 2 \Longrightarrow y^2 \equiv 20 \equiv 0 \Longrightarrow y \equiv 0, 3 \pmod 5 \quad x \equiv 3 \Longrightarrow y^2 \equiv 43 \equiv 3 \Longrightarrow \text{no solutions} \quad x \equiv 4 \Longrightarrow y^2 \equiv 84 \equiv 4 \Longrightarrow y \equiv 2, 3 \pmod 5 \quad x = \infty \Longrightarrow y = \infty .$$

The points on E are
(0, 2), (0, 3), (1, 2), (1, 3), (2, 0), (4, 2), (4, 3), (

The addition of points on an elliptic curve mod p is done via the same formulas as given previously, except that a rational number a/b must be treated as ab^{-1}, where $b^{-1}b \equiv 1 \pmod{p}$. This requires that gcd(b, p)=1.

More generally, it is possible to develop a theory of elliptic curves mod n for any integer n. In this case, when we encounter a fraction a/b, we need to have gcd(b, n)=1. The situations where this fails form the key to using elliptic curves for factorization, as we'll see in Section 21.3. There are various technical problems in the general theory that arise when $1<\gcd(b, n)<n$, but the

method to overcome these will not be needed in the following. For details on how to treat this case, see [Washington]. For our purposes, when we encounter an elliptic curve mod a composite n, we can pretend n is prime. If something goes wrong, we usually obtain useful information about n, for example its factorization.

Example

Let's compute (1, 2)+(4, 3) on the curve just considered. The slope is

$$m \equiv 3-24-1 \equiv 2 \pmod 5.$$

Therefore,

$$x3 \equiv m2-x1-x2 \equiv 22-1-4 \equiv 4 \pmod 5$$
$$y3 \equiv m(x1-x3)-y1 \equiv 2(1-4)-2 \equiv 2 \pmod 5.$$

This means that

$$(1, 2)+(4, 3)=(4, 2).$$

Example

Here is a somewhat larger example. Let $n=2773$. Let

$$E: y^2 \equiv x^3+4x+4 \pmod{2773}, \text{ and } P=(1, 3).$$

Let's compute $2P=P+P$. To get the slope of the tangent line, we differentiate implicitly and evaluate at $(1, 3)$:

$$2ydy=(3x^2+4)dx \Rightarrow \frac{dy}{dx}=\frac{7}{6}.$$

But we are working mod 2773. Using the extended Euclidean algorithm (see Section 3.2), we find that $2311 \cdot 6 \equiv 1 \pmod{2773}$, so we can replace $1/6$ by 2311. Therefore,

$$m \equiv \frac{7}{6} \equiv 7 \times 2311 \equiv 2312 \pmod{2773}.$$

The formulas yield

$$x_3 \equiv 23122 - 1 - 1 \equiv 1771 \pmod{2773}$$
$$y_3 \equiv 2312(1 - 1771) - 3 \equiv 705 \pmod{2773}.$$

The final answer is

$$2P = P + P = (1771, 705).$$

Now that we're done with the example, we mention that 2773 is not prime. When we try to calculate 3P in Section 21.3, we'll obtain the factorization of 2773.

21.2.1 Number of Points Mod *p*

Let $E: y^2 \equiv x^3 + bx + c \pmod{p}$ be an elliptic curve, where $p \geq 5$ is prime. We can list the points on E by letting $x = 0, 1, \ldots, p-1$ and seeing when $x^3 + bx + c$ is a square mod p. Since half of the nonzero numbers are squares mod p, we

expect that x3+bx+c will be a square approximately half the time. When it is a nonzero square, there are two square roots: y and −y. Therefore, approximately half the time we get two values of y and half the time we get no y. Therefore, we expect around p points. Including the point ∞, we expect a total of approximately p+1 points. In the 1930s, H. Hasse made this estimate more precise.

Hasse's Theorem

Suppose E (mod p) has N points. Then

$$|N-p-1| < 2p.$$

The proof of this theorem is well beyond the scope of this book (for a proof, see [Washington]). It can also be

shown that whenever N and p satisfy the inequality of the theorem, there is an elliptic curve E mod p with exactly N points.

If p is large, say around 1020, it is infeasible to count the points on an elliptic curve by listing them. More sophisticated algorithms have been developed by Schoof, Atkin, Elkies, and others to deal with this problem. See the Sage Appendix.

21.2.2 Discrete Logarithms on Elliptic Curves

Recall the classical discrete logarithm problem: We know that $x \equiv g^k \pmod{p}$ for some k, and we want to find k. There is an elliptic curve version:

Suppose we have points A, B on an elliptic curve E and we know that $B=kA(=A+A+\cdots+A)$ for some integer k. We want to find k. This might not look like a logarithm problem, but it is clearly the analog of the classical discrete logarithm problem. Therefore, it is called the **discrete logarithm problem** for elliptic curves.

There is no good general attack on the discrete logarithm problem for elliptic curves. There is an analog of the Pohlig-Hellman attack that works in some situations. Let E be an elliptic curve mod a prime p and let n be the smallest integer such that $nA=\infty$. If n has only small prime factors, then it is possible to calculate the discrete logarithm k mod

the prime powers dividing n and then use the Chinese remainder theorem to find k (see Exercise 25). The Pohlig-Hellman attack can be thwarted by choosing E and A so that n has a large prime factor.

There is no replacement for the index calculus attack described in Section 10.2. This is because there is no good analog of "small." You might try to use points with small coordinates in place of the "small primes," but this doesn't work. When you factor a number by dividing off the prime factors one by one, the quotients get smaller and smaller until you finish. On an elliptic curve, you could have a point with fairly small coordinates, subtract off a small

point, and end up with a point with large coordinates (see Computer Problem 5). So there is no good way to know when you are making progress toward expressing a point in terms of the factor base of small points.

The Baby Step, Giant Step attack on discrete logarithms works for elliptic curves (Exercise 13(b)), although it requires too much memory to be practical in most situations. For other attacks, see [Blake et al.] and [Washington].

21.2.3 Representing Plaintext

In most cryptographic systems, we must have a method for mapping our original

message into a numerical value upon which we can perform mathematical operations. In order to use elliptic curves, we need a method for mapping a message onto a point on an elliptic curve. Elliptic curve cryptosystems then use elliptic curve operations on that point to yield a new point that will serve as the ciphertext.

The problem of encoding plaintext messages as points on an elliptic curve is not as simple as it was in the conventional case. In particular, there is no known polynomial time, deterministic algorithm for writing down points on an arbitrary elliptic curve $E \pmod{p}$. However, there are fast probabilistic methods for finding points,

and these can be used for encoding messages. These methods have the property that with small probability they will fail to produce a point. By appropriately choosing parameters, this probability can be made arbitrarily small, say on the order of 1/230.

Here is one method, due to Koblitz. The idea is the following. Let E:y2 ≡ x3+bx +c (mod p) be the elliptic curve. The message m (already represented as a number) will be embedded in the x-coordinate of a point. However, the probability is only about 1/2 that m3+bm+c is a square mod p. Therefore, we adjoin a few bits at the end of m and adjust them until we get a number x such that x3+bx+c is a square mod p.

More precisely, let K be a large integer so that a failure rate of 1/2K is acceptable when trying to encode a message as a point. Assume that m satisfies $(m+1)K<p$. The message m will be represented by a number $x=mK+j$, where $0 \leq j<K$. For $j=0, 1, \ldots, K-1$, compute x^3+bx+c and try to calculate the square root of $x^3+bx+c \pmod{p}$. For example, if $p \equiv 3 \pmod{4}$, the method of Section 3.9 can be used. If there is a square root y, then we take $Pm=(x, y)$; otherwise, we increment j by one and try again with the new x. We repeat this until either we find a square root or $j=K$. If j ever equals K, then we fail to map a message to a point. Since x^3+bx+c is a square approximately half of the time, we have about a 1/2K chance of failure.

In order to recover the message from the point Pm=(x, y) we simply calculate m by

$$m=\lfloor x/K \rfloor,$$

where $\lfloor x/K \rfloor$ denotes the greatest integer less than or equal to x/K.

Example

Let p=179 and suppose that our elliptic curve is y2=x3+2x+7. If we are satisfied with a failure rate of 1/210, then we may take K=10. Since we need (m +1)K<179, we need $0 \leq m \leq 16$. Suppose our message is m=5. We consider x of the form mK+j=50+j. The possible choices for x are 50, 51, … , 59. For x=51 we get $x3+2x+7 \equiv 121 \pmod{179}$, and $112 \equiv 121 \pmod{179}$.

Thus, we represent the message m=5 by the point Pm=(51, 11). The message m can be recovered by m=[51/10]=5.

21.3 Factoring with Elliptic Curves

Suppose n=pq is a number we wish to factor. Choose a random elliptic curve mod n and a point on the curve. In practice, one chooses several (around 14 for numbers around 50 digits; more for larger integers) curves with points and runs the algorithm in parallel.

How do we choose the curve? First, choose a point P and a coefficient b. Then choose c so that P lies on the curve $y2=x3+bx+c$. This is much more efficient than choosing b and c and then trying to find a point.

For example, let n=2773. Take P=(1, 3) and b=4. Since we want $32 \equiv 13+4\cdot 1+c$, we take c=4. Therefore, our curve is

$$E: y^2 \equiv x^3+4x+4 \pmod{2773}.$$

We calculated 2P=(1771, 705) in a previous example. Note that during the calculation, we needed to find $6^{-1} \pmod{2773}$. This required that $\gcd(6, 2773)=1$ and used the extended Euclidean algorithm, which was essentially a gcd calculation.

Now let's calculate 3P=2P+P. The line through the points 2P=(1771, 705) and P=(1, 3) has slope 702/1770. When we try to invert 1770 mod 2773, we find that $\gcd(1770, 2773)=59$, so we cannot do this. So what do we do? Our original goal was to factor 2773, so we don't

need to do anything more. We have found the factor 59, which yields the factorization $2773=59\cdot 47$.

Here's what happened. Using the Chinese remainder theorem, we can regard E as a pair of elliptic curves, one mod 59 and the other mod 47. It turns out that $3P=\infty \pmod{59}$, while $4P=\infty \pmod{47}$. Therefore, when we tried to compute 3P, we had a slope that was infinite mod 59 but finite mod 47. In other words, we had a denominator that was 0 mod 59 but nonzero mod 47. Taking the gcd allowed us to isolate the factor 59.

The same type of idea is the basis for many factoring algorithms. If $n=pq$, you cannot separate p and q as long as

they behave identically. But if you can find something that makes them behave slightly differently, then they can be separated. In the example, the multiples of P reached ∞ faster mod 59 than mod 47. Since in general the primes p and q should act fairly independently of each other, one would expect that for most curves E (mod pq) and points P, the multiples of P would reach ∞ mod p and mod q at different times. This will cause the gcd to find either p or q.

Usually, it takes several more steps than 3 or 4 to reach ∞ mod p or mod q. In practice, one multiplies P by a large number with many small prime factors, for example, 10000!. This can be done via successive doubling (the additive

analog of successive squaring; see Exercise 21). The hope is that this multiple of P is ∞ either mod p or mod q. This is very much the analog of the $p-1$ method of factoring. However, recall that the $p-1$ method (see Section 9.4) usually doesn't work when $p-1$ has a large prime factor. The same type of problem could occur in the elliptic curve method just outlined when the number m such that mP equals ∞ has a large prime factor. If this happens (so the method fails to produce a factor after a while), we simply change to a new curve E. This curve will be independent of the previous curve and the value of m such that $mP=\infty$ should have essentially no relation to the previous m. After several tries (or if several curves are

treated in parallel), a good curve is often found, and the number n=pq is factored. In contrast, if the p−1 method fails, there is nothing that can be changed other than using a different factorization method.

Example

We want to factor n=455839. Choose

$$E: y^2 \equiv x^3+5x-5,\ P=(1,\ 1).$$

Suppose we try to compute 10!P. There are many ways to do this. One is to compute 2!P, 3!P=3(2!P), 4!P=4(3!P), …. If we do this, everything is fine through 7!P, but 8!P requires inverting $599 \pmod{n}$. Since gcd(599, n)=599, we can factor n as 599×761.

Let's examine this more closely. A computation shows that $E \pmod{599}$ has $640=27\times5$ points and $E \pmod{761}$ has $777=3\times7\times37$ points. Moreover, 640 is the smallest positive m such that $mP=\infty$ on $E \pmod{599}$, and 777 is the smallest positive m such that $mP=\infty$ on $E \pmod{761}$. Since 8! is a multiple of 640, it is easy to see that $8!P=\infty$ on $E \pmod{599}$, as we calculated. Since 8! is not a multiple of 777, it follows that $8!P \neq \infty$ on $E \pmod{761}$. Recall that we obtain ∞ when we divide by 0, so calculating 8!P asked us to divide by $0 \pmod{599}$. This is why we found the factor 599.

For more examples, see Examples 45 and 46 in the Computer Appendices.

In general, consider an elliptic curve E

(mod p) for some prime p. The smallest positive m such that $mP = \infty$ on this curve divides the number N of points on E (mod p) (if you know group theory, you'll recognize this as a corollary of Lagrange's theorem), so $NP = \infty$. Quite often, m will be N or a large divisor of N. In any case, if N is a product of small primes, then B! will be a multiple of N for a reasonably small value of B. Therefore, $B!P = \infty$.

A number that has only small prime factors is called **smooth**. More precisely, if all the prime factors of an integer are less than or equal to B, then it is called **B-smooth**. This concept played a role in the $x2 \equiv y2$ method and the p−1 factoring method (Section 9.4),

and the index calculus attack on discrete logarithms (Section 10.2).

Recall from Hasse's theorem that N is an integer near p. It is possible to show that the density of smooth integers is large enough (we'll leave *small* and *large* undefined here) that if we choose a random elliptic curve $E \pmod p$, then there is a reasonable chance that the number N is smooth. This means that the elliptic curve factorization method should find p for this choice of the curve. If we try several curves $E \pmod n$, where $n=pq$, then it is likely that at least one of the curves $E \pmod p$ or $E \pmod q$ will have its number of points being smooth.

In summary, the advantage of the

elliptic curve factorization method over the p−1 method is the following. The p −1 method requires that p−1 is smooth. The elliptic curve method requires only that there are enough smooth numbers near p so that at least one of some randomly chosen integers near p is smooth. This means that elliptic curve factorization succeeds much more often than the p−1 method.

The elliptic curve method seems to be best suited for factoring numbers of medium size, say around 40 or 50 digits. These numbers are no longer used for the security of factoring-based systems such as RSA, but it is sometimes useful in other situations to have a fast factorization method for such numbers.

Also, the elliptic curve method is effective when a large number has a small prime factor, say of 10 or 20 decimal digits. For large numbers where the prime factors are large, the quadratic sieve and number field sieve are superior (see Section 9.4).

21.3.1 Singular Curves

In practice, the case where the cubic polynomial x^3+bx+c has multiple roots rarely arises. But what happens if it does? Does the factorization algorithm still work? The discriminant $4b^3+27c^2$ is zero if and only if there is a multiple root (this is the cubic analog of the fact that ax^2+bx+c has a double root if and only if $b^2-4ac=0$). Since we are working mod $n=pq$, the result says that

there is a multiple root mod n if and only if the discriminant is 0 mod n. Since n is composite, there is also the intermediate case where the gcd of n and the discriminant is neither 1 nor n. But this gives a nontrivial factor of n, so we can stop immediately in this case.

Example

Let's look at an example:

$$y2=x3−3x+2=(x−1)2(x+2).$$

Given a point P=(x, y) on this curve, we associate the number

$$(y+3(x−1))/(y−3(x−1)).$$

It can be shown that adding the points on the curve corresponds to multiplying the corresponding numbers. The

formulas still work, as long as we don't use the point (1, 0). Where does this come from? The two lines tangent to the curve at (1, 0) are $y+3(x-1)=0$ and $y-3(x-1)=0$. This number is simply the ratio of these two expressions.

Since we need to work mod n, we give an example mod 143. We choose 143 since 3 is a square mod 143; in fact, $822 \equiv 3 \pmod{143}$. If this were not the case, things would become more technical with this curve. We could easily rectify the situation by choosing a new curve.

Consider the point $P=(-1, 2)$ on $y2=x3-3x+2 \pmod{143}$. Look at its multiples:

P=(−1, 2), 2P=(2, 141), 3P=(112, 101), 4P=(10, 20).

When trying to compute 5P, we find the factor 11 of 143.

Recall that we are assigning numbers to each point on the curve, other than (1,1). Since we are working mod 143, we use 82 in place of 3. Therefore, the number corresponding to (−1, 2) is $(2+82(-1-1))/(2-82(-1-1)) = 80 \pmod{143}$. We can compute the numbers for all the points above:

P↔80, 2P↔108, 3P↔60, 4P↔81.

Let's compare with the powers of 80 mod 143:

$80^1 \equiv 80,\ 80^2 \equiv 108,\ 80^3 \equiv 60,\ 80^4 \equiv 81,\ 80^5 \equiv 45.$

We get the same numbers. This is simply the fact mentioned previously

that the addition of points on the curve corresponds to multiplication of the corresponding numbers. Moreover, note that $45 \equiv 1 \pmod{11}$, but not mod 13. This corresponds to the fact that 5 times the point $(-1, 2)$ is ∞ mod 11 but not mod 13. Note that 1 is the multiplicative identity for multiplication mod 11, while ∞ is the additive identity for addition on the curve.

It is easy to see from the preceding that factorization using the curve $y2=x3-3x+2$ is essentially the same as using the classical $p-1$ factorization method (see Section 9.4).

In the preceding example, the cubic equation had a double root. An even worse possibility is the cubic having a

triple root. Consider the curve

y2=x3.

To a point (x, y) ≠ (0, 0) on this curve, associate the number x/y. Let's start with the point P=(1, 1) and compute its multiples:

P=(1, 1), 2P=(14, 18), 3P=(19, 127), … , mP=(1m2, 1m3).

Note that the corresponding numbers x/y are 1, 2, 3, … , m. Adding the points on the curve corresponds to adding the numbers x/y.

If we are using the curve y2=x3 to factor n, we need to change the points mP to integers mod n, which requires finding inverses for m2 and m3 mod n. This is done by the extended Euclidean algorithm, which is essentially a gcd

computation. We find a factor of n when gcd(m, n) ≠ 1. Therefore, this method is essentially the same as computing in succession
gcd(2, n), gcd(3, n), gcd(4, n), … until a factor is found. This is a slow version of trial division, the oldest factorization technique known. Of course, in the elliptic curve factorization algorithm, a large multiple (B!)P of P is usually computed. This is equivalent to factoring by computing gcd(B!, n), a method that is often used to test for prime factors up to B.

In summary, we see that the p−1 method and trial division are included in the elliptic curve factorization algorithm if we allow singular curves.

21.4 Elliptic Curves in Characteristic 2

Many applications use elliptic curves mod 2, or elliptic curves defined over the finite fields GF(2n) (these are described in Section 3.11). This is often because mod 2 adapts well to computers. In 1999, NIST recommended 15 elliptic curves for cryptographic uses (see [FIPS 186-2]). Of these, 10 are over finite fields GF(2n).

If we're working mod 2, the equations for elliptic curves need to be modified slightly. There are many reasons for this. For example, the derivative of y2 is

$2yy' = 0$, since 2 is the same as 0. This means that the tangent lines we compute are vertical, so $2P = \infty$ for all points P. A more sophisticated explanation is that the curve $y^2 \equiv x^3 + bx + c \pmod 2$ has singularities (points where the partial derivatives with respect to x and y simultaneously vanish).

The equations we need are of the form

$$E: y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$

where $a_1, \ldots, a_6$ are constants. The addition law is slightly more complicated. We still have three points adding to infinity if and only if they lie on a line. Also, the lines through ∞ are vertical. But, as we'll see in the following example, finding $-P$ from P is not the same as before.

Example

Let E:$y^2+y \equiv x^3+x \pmod 2$. As before, we can list the points on E:

$(0, 0), (0, 1), (1, 0), (1, 1), \infty$.

Let's compute $(0, 0)+(1, 1)$. The line through these two points is $y=x$. Substituting into the equation for E yields $x^2+x \equiv x^3+x$, which can rewritten as $x^2(x+1) \equiv 0$. The roots are $x=0, 0, 1 \pmod 2$. Therefore, the third point of intersection also has $x=0$. Since it lies on the line $y=x$, it must be $(0, 0)$. (This might be puzzling. What is happening is that the line is tangent to E at $(0, 0)$ and also intersects E in the point $(1, 1)$.) As before, we now have

$(0, 0)+(0, 0)+(1, 1)=\infty$.

To get $(0, 0)+(1, 1)$ we need to compute $\infty - (0, 0)$. This means we need to find P such that $P+(0, 0)=\infty$. A line through ∞ is still a vertical line. In this case, we need one through $(0, 0)$, so we take $x=0$. This intersects E in the point $P=(0, 1)$. We conclude that $(0, 0)+(0, 1)=\infty$. Putting everything together, we see that

$$(0, 0)+(1, 1)=(0, 1).$$

In most applications, elliptic curves mod 2 are not large enough. Therefore, elliptic curves over finite fields are used. For an introduction to finite fields, see Section 3.11. However, in the present section, we only need the field GF(4), which we now describe.

Let

GF(4)={0, 1, ω, ω2},

with the following laws:

1. 0+x=x for all x.
2. x+x=0 for all x.
3. 1·x=x for all x.
4. 1+ω=ω2.
5. Addition and multiplication are commutative and associative, and the distributive law holds: x(y +z)=xy+xz for all x, y, z.

Since

ω3=ω·ω2=ω·(1+ω)=ω+ω2=ω+(1+ω)=1,

we see that ω2 is the multiplicative inverse of ω. Therefore, every nonzero element of GF(4) has a multiplicative inverse.

Elliptic curves with coefficients in finite

fields are treated just like elliptic curves with integer coefficients.

Example

Consider

$$E : y2+xy=x3+\omega,$$

where $\omega \in GF(4)$ is as before. Let's list the points of E with coordinates in GF(4):

$$x=0 \Rightarrow y2=\omega \Rightarrow y=\omega 2 x=1 \Rightarrow y2+y=1+\omega=\omega 2 \Rightarrow \text{ no solutions} x=\omega \Rightarrow y2+\omega y=\omega 2 \Rightarrow y=1,$$

The points on E are therefore

$$(0, \omega 2), (\omega, 1), (\omega, \omega 2), \infty .$$

Let's compute $(0, \omega 2)+(\omega, \omega 2)$. The line through these two points is $y=\omega 2$. Substitute this into the equation for E:

$$\omega 4+\omega 2x=x3+\omega,$$

which becomes $x3+\omega 2x=0$. This has the roots $x=0, \omega, \omega$. The third point of intersection of the line and E is therefore $(\omega, \omega 2)$, so

$$(0, \omega 2)+(\omega, \omega 2)+(\omega, \omega 2)=\infty .$$

We need $-(\omega, \omega 2)$, namely the point P with $P+(\omega, \omega 2)=\infty$. The vertical line $x=\omega$ intersects E in $P=(\omega, 1)$, so

$$(0, \omega 2)+(\omega, \omega 2)=(\omega, 1).$$

For cryptographic purposes, elliptic curves are used over fields GF(2n) with n large, say at least 150.

21.5 Elliptic Curve Cryptosystems

Elliptic curve versions exist for many cryptosystems, in particular those involving discrete logarithms. An advantage of elliptic curves compared to working with integers mod p is the following. In the integers, it is possible to use the factorization of integers into primes (especially small primes) to attack the discrete logarithm problem. This is known as the index calculus and is described in Section 10.2. There seems to be no good analog of this method for elliptic curves. Therefore, it is possible to use smaller primes, or smaller finite fields, with elliptic curves

and achieve a level of security comparable to that for much larger integers mod p. This allows great savings in hardware implementations, for example.

In the following, we describe three elliptic curve versions of classical algorithms. Here is a general procedure for changing a classical system based on discrete logarithms into one using elliptic curves:

Notes:

1. The elliptic curve is an elliptic curve mod some prime, so n, the number of points on the curve, including ∞, is finite.

2. Addition and subtraction of points on an elliptic curve are of equivalent complexity (if $Q=(x, y)$, then $-Q=(x, -y)$ and $P-Q$ is computed as $P+(-Q)$), but multiplication mod p is much easier than division mod p (via the extended Euclidean algorithm). Both mod p operations are usually simpler than the elliptic curve operations.

3. The elliptic curve discrete log problem is believed to be harder than the mod p discrete log problem.

4. If we fix a number m and look at the set of all integers mod m, then the analogues of the above are: addition mod m, the additive identity 0, subtraction mod m, multiplying an integer times a number mod m (that is, $ka=a+a+\cdots+a \pmod m$), m= the number of integers mod m, the relation $ma \equiv 0 \pmod m$, and the additive discrete log problem: Solve $ka \equiv b \pmod m$ for k, which can be done easily via the Extended Euclidean algorithm. This shows that the difficulty of a discrete log problem depends on the binary operation.

21.5.1 An Elliptic Curve ElGamal Cryptosystem

We recall the non-elliptic curve version.

Alice wants to send a message x to Bob, so Bob chooses a large prime p and an integer α mod p. He also chooses a secret integer s and computes $\beta \equiv \alpha s \pmod{p}$. Bob makes p, α, β public and keeps s secret. Alice chooses a random k and computes y1 and y2, where

$$y1 \equiv \alpha k \text{ and } y2 \equiv x\beta k \pmod{p}.$$

She sends (y1, y2) to Bob, who then decrypts by calculating

$$x \equiv y2y1{-}s \pmod{p}.$$

Now we describe the elliptic curve version. Bob chooses an elliptic curve E (mod p), where p is a large prime. He chooses a point α on E and a secret integer s. He computes

$$\beta = s\alpha \ (=\alpha+\alpha+\cdots+\alpha).$$

The points α and β are made public, while s is kept secret. Alice expresses her message as a point x on E (see Section 21.5). She chooses a random integer k, computes

$$y1=k\alpha \text{ and } y2=x+k\beta,$$

and sends the pair y1, y2 to Bob. Bob decrypts by calculating

$$x=y2-sy1.$$

A more workable version of this system is due to Menezes and Vanstone. It is described in [Stinson1, p. 189].

Example

We must first generate a curve. Let's use the prime p=8831, the point G=(x, y)=(4, 11), and b=3. To make G

lie on the curve $y2 \equiv x3+bx+c \pmod p$, we take c=45. Alice has a message, represented as a point Pm=(5, 1743), that she wishes to send to Bob. Here is how she does it.

Bob has chosen a secret random number sB=3 and has published the point sBG=(413, 1808).

Alice downloads this and chooses a random number k=8. She sends Bob kG=(5415, 6321) and Pm +k(sBG)=(6626, 3576). He first calculates sB(kG)=3(5415, 6321)=(673, 146). He now subtracts this from (6626, 3576):

(6626, 3576)−(673, 146)=(6626, 3576)+(673, −146)=(5, 1743).

Note that we subtracted points by using

the rule $P-Q=P+(-Q)$ from Section 21.1.

For another example, see Example 47 in the Computer Appendices.

21.5.2 Elliptic Curve Diffie-Hellman Key Exchange

Alice and Bob want to exchange a key. In order to do so, they agree on a public basepoint G on an elliptic curve $E: y^2 \equiv x^3+bx+c \pmod{p}$. Let's choose $p=7211$ and $b=1$ and $G=(3, 5)$. This forces us to choose $c=7206$ in order to have the point on the curve. Alice chooses NA randomly and Bob chooses NB randomly. Let's suppose $NA=12$ and $NB=23$. They keep these private to themselves but publish NAG and NBG.

In our case, we have

$NAG=(1794, 6375)$ and $NBG=(3861, 1242)$.

Alice now takes NBG and multiplies by NA to get the key:

$NA(NBG)=12(3861, 1242)=(1472, 2098)$.

Similarly, Bob takes NAG and multiplies by NB to get the key:

$NB(NAG)=23(1794, 6375)=(1472, 2098)$.

Notice that they have the same key.

For another example, see Example 48 in the Computer Appendices.

21.5.3 ElGamal Digital Signatures

There is an elliptic curve analog of the

procedure described in Section 13.2. A few modifications are needed to account for the fact that we are working with both integers and points on an elliptic curve.

Alice wants to sign a message m (which might actually be the hash of a long message). We assume m is an integer. She fixes an elliptic curve E (mod p), where p is a large prime, and a point A on E. We assume that the number of points N on E has been calculated and assume $0 \leq m < N$ (if not, choose a larger p). Alice also chooses a private integer a and computes B=aA. The prime p, the curve E, the integer n, and the points A and B are made public. To sign the message, Alice does the following:

1. Chooses a random integer k with $1 \leq k < N$ and $\gcd(k, N)=1$, and computes $R=kA=(x, y)$
2. Computes $s \equiv k-1(m-ax) \pmod N$
3. Sends the signed message (m, R, s) to Bob

Note that R is a point on E, and m and s are integers.

Bob verifies the signature as follows:

1. Downloads Alice's public information p, E, A, B
2. Computes $V1=xB+sR$ and $V2=mA$
3. Declares the signature valid if $V1=V2$

The verification procedure works because

$V1=xB+sR=xaA+k-1(m-ax)(kA)=xaA+(m-ax)A=mA=V2.$

There is a subtle point that should be mentioned. We have used $k-1$ in this

verification equation as the integer mod N satisfying $k^{-1}k \equiv 1 \pmod{N}$. Therefore, $k^{-1}k$ is not 1, but rather an integer congruent to 1 mod N. So $k^{-1}k = 1 + tN$ for some integer t. It can be shown that $NA = \infty$. Therefore,

$$k^{-1}kA = (1+tN)A = A + t(NA) = A + t\infty = A.$$

This shows that k^{-1} and k cancel each other in the verification equation, as we implicitly assumed above.

The classical ElGamal scheme and the present elliptic curve version are analogs of each other. The integers mod p are replaced with the elliptic curve E, and the number $p-1$ becomes N. Note that the calculations in the classical scheme work with integers that are nonzero mod p, and there are $p-1$ such

congruence classes. The elliptic curve version works with points on the elliptic curve that are multiples of A, and the number of such points is a divisor of N.

The use of the x-coordinate of R in the elliptic version is somewhat arbitrary. Any method of assigning integers to points on the curve would work. Using the x-coordinate is an easy choice. Similarly, in the classical ElGamal scheme, the use of the integer r in the mod p−1 equation for s might seem a little unnatural, since r was originally defined mod p. However, any method of assigning integers to the integers mod p would work (see Exercise 16 in Chapter 13). The use of r itself is an easy choice.

There is an elliptic curve version of the

Digital Signature Algorithm that is similar to the preceding (Exercise 24).

21.6 Exercises

1.

    1. Let $x3+ax2+bx+c$ be a cubic polynomial with roots $r1, r2, r3$. Show that $r1+r2+r3=−a$.

    2. Write $x=x1−a/3$. Show that

       $$x3+ax2+bx+c=x13+b′x1+c′,$$

       with $b′=b−(1/3)a2$ and $c′=c−(1/3)ab+(2/27)a3$. (Remark: This shows that a simple change of variables allows us to consider the case where the coefficient of $x2$ is 0.)

2. Let E be the elliptic curve $y2 \equiv x3−x+4 \pmod 5$.

    1. List the points on E (don't forget ∞).

    2. Evaluate the elliptic curve addition $(2, 0)+(4, 3)$.

3.

    1. List the points on the elliptic curve

E:$y^2 \equiv x^3 - 2 \pmod 7$.

2. Find the sum $(3, 2)+(5, 5)$ on E.

3. Find the sum $(3, 2)+(3, 2)$ on E.

4. Let E be the elliptic curve $y^2 \equiv x^3+x+2 \pmod{13}$.

    1. Evaluate $(1, 2)+(2, 5)$.

    2. Evaluate $2(1, 2)$.

    3. Evaluate $(1, 2)+\infty$.

5.

    1. Find the sum of the points (1, 2) and (6,3) on the elliptic curve $y^2 \equiv x^3+3 \pmod 7$.

    2. Eve tries to find the sum of the points (1,2) and (6,3) on the elliptic curve $y^2 \equiv x^3+3 \pmod{35}$. What information does she obtain?

6. Show that if $P=(x, 0)$ is a point on an elliptic curve, then $2P=\infty$.

7. Find an elliptic curve mod 101 such that (43, 21) is a point on the curve.

8. The point (3, 5) lies on the elliptic curve $y2=x3-2$ defined over the rational numbers. use the addition law to find another point with positive rational coordinates that lies on this curve.

9.
    1. Show that $Q=(2, 3)$ on $y2=x3+1$ satisfies $6Q=\infty$. (Hint: Compute $3Q$, then use Exercise 6.)

    2. Your computations in (a) probably have shown that $2Q \neq \infty$ and $3Q \neq \infty$. Use this to show that the points ∞, Q, 2Q, 3Q, 4Q, 5Q are distinct.

10.
    1. Factor $n=35$ by the elliptic curve method by using the elliptic curve $y2 \equiv x3+26$ and calculating 3 times the point $P=(10, 9)$.

    2. Factor $n=35$ by the elliptic curve method by using the elliptic curve $y2 \equiv x3+5x+8$ and the point $P=(1, 28)$.

11. Suppose you want to factor a composite integer n by using the elliptic curve method. You start with the curve $y2=x3-4x \pmod n$ and the point (2, 0). Why will this not yield the factorization of n?

12. Devise an analog of the procedure in Exercise 11(a) in Chapter 10 that uses elliptic curves.

13. Let p=999983. The elliptic curve E:$y^2 \equiv x^3+1 \pmod{p}$ has 999984 points. Suppose you are given points P and Q on E and are told that there is an integer k such that Q=kP.

    1. Describe a birthday attack that is expected to find k.

    2. Describe how the Baby Step, Giant Step method (see Section 10.2) finds k.

14. Let P and Q be points on an elliptic curve E. Peggy claims that she knows an integer k such that kP=Q and she wants to convince Victor that she knows k without giving Victor any information about k. They perform a zero-knowledge protocol. The first step is the following:

    1. Peggy chooses a random integer r1 and lets r2=k−r1. She computes X1=r1P and X2=r2P and sends them to Victor.

       Give the remaining steps. Victor wants to be at least 99% sure that Peggy knows k. (Technical note: You may regard r1 and r2 as numbers mod n, where $nP=\infty$. Without congruences, Victor obtains

some information about the size of k.

Nontechnical note: The “Technical note” may be ignored when solving the problem.)

15. Find all values of y mod 35 such that (1, y) is a point on the curve $y2 \equiv x3+3x+12 \pmod{35}$.

16. Suppose n is a product of two large primes and let $E: y2 \equiv x3+bx+c \pmod{n}$. Bob wants to find some points on E.

    1. Bob tries choosing a random x, computing x3+bx+c, and finding the square root of this number mod n, when the square root exists. Why will this strategy probably fail if Bob does not know p and q?

    2. Suppose Bob knows p and q. Explain how Bob can use the method of part (a) successfully? (Hint: He needs to use the Chinese Remainder Theorem.)

17. Show that if P, Q, R are points on an elliptic curve, then

    $P+Q+R=\infty \Leftrightarrow$ P, Q, R are collinear.

18.

1. Eve is trying to find an elliptic curve discrete log: She has points A and B on an elliptic curve E such that B=kA for some k. There are approximately 1020 points on E, so assume that $1 \le k \le 1020$. She makes two lists and looks for a match. The first list is jA for N randomly chosen values of j. The second is $B-\ell A$ for N randomly chosen values of ℓ. How big should N be so that there is a good chance of a match?

2. Give a classical (that is, not elliptic curve) version of the procedure in part (a).

19. Let P be a point on the elliptic curve E mod a prime p.

    1. Show that there are only finitely many points on E, so P has only finitely many distinct multiples.

    2. Show that there are integers i, j with i>j such that iP=jP. Conclude that $(i-j)P=\infty$.

    3. The smallest positive integer k such that $kP=\infty$ is called the **order** of P. Let m be an integer such that $mP=\infty$. Show that k divides m. (Hint: Imitate the proof of

Exercise 53(c, d) in Chapter 3.)

4. (for those who know some group theory) Use Lagrange's theorem from group theory to show that the number of points on E is a multiple of the order of P. (Combined with Hasse's theorem, this gives a way of finding the number of points on E. See Computer Problems 1 and 4.)

20. Let P be a point on the elliptic curve E. Suppose you know a positive integer k such that $kP = \infty$. You want to prove (or disprove) that k is the order of P.

    1. Show that if $(k/p)P = \infty$ for some prime factor p of k, then k is not the order of P.

    2. Suppose $m|k$ and $1 \le m < k$. Show that $m|(k/p)$ for some prime divisor p of k.

    3. Suppose that $(k/p)P \neq \infty$ for each prime factor of k. Use Exercise 11(c) to show that the order of P is k. (Compare with Exercise 54 in Chapter 3. For an example, see Computer Problem 4.)

21.

1. Let $x=b_1b_2 \ldots b_w$ be an integer written in binary. Let P be a point on the elliptic curve E. Perform the following procedure:

    1. Start with $k=1$ and $S_1=\infty$.

    2. If $b_k=1$, let $R_k=S_k+P$. If $b_k=0$, let $R_k=S_k$.

    3. Let $S_{k+1}=2R_k$.

    4. If $k=w$, stop. If $k<w$, add 1 to k and go to step 2.

    Show that $R_w=xP$. (Compare with Exercise 56(a) in Chapter 3.)

2. Let x be a positive integer and let P be a point on an elliptic curve. Show that the following procedure computes xP.

    1. Start with $a=x$, $B=\infty$, $C=P$.

    2. If a is even, let $a=a/2$, and let $B=B$, $C=2C$.

    3. If a is odd, let $a=a-1$, and let $B=B+C$, $C=C$.

4. If $a \neq 0$, go to step 2.

5. Output B.

(Compare with Exercise 56(b) in Chapter 3.)

22. Let E be an elliptic curve mod n (where n is some integer) and let P and Q be points on E with 2P=Q. The curve E and the point Q are public and are known to everyone. The point P is secret. Peggy wants to convince Victor that she knows P. They do the following procedure:

    1. Peggy chooses a random point R1 on E and lets R2=P−R1.

    2. Peggy computes H1=2R1 and H2=2R2 and sends H1, H2 to Victor.

    3. Victor checks that H1+H2=Q.

    4. Victor makes a request and Peggy responds.

    5. Victor now does something else.

    6. They repeat steps 1 through 5 several times.

1. Describe what is done in steps 4 and 5.

2. Give a classical (non-elliptic curve) version of this protocol that yields a zero-knowledge proof that Peggy knows a solution x to $x2 \equiv s \bmod n$.

23. Let E be an elliptic curve mod a large prime, let N be the number of points on E, and let P and Q be points on E. Peggy claims to know an integer s such that sP=Q. She wants to prove this to Victor by the following procedure. Victor knows E, P, and Q, but he does not know s and should receive no information about s.

    1. Peggy chooses a random integer r1 mod N and lets $r2 \equiv s - r1 \pmod N$. (Don't worry about why it's mod N. It's for technical reasons.)

    2. Peggy computes Y1=r1P and Y2=r2P and sends Y1 and Y2 to Victor.

    3. Victor checks something.

    4. Victor randomly chooses i=1 or 2 and asks Peggy for ri.

    5. Peggy sends ri to Victor.

6. Victor checks something.

7. Step (7).

1. What does Victor check in step (3)?

2. What does Victor check in step (6)?

3. What should step (7) be if Victor wants to be at least 99.9% sure that Peggy knows s?

24. Here is an elliptic curve version of the Digital Signature Algorithm. Alice wants to sign a message m, which is an integer. She chooses a prime p and an elliptic curve $E \pmod p$. The number of points n on E is computed and a large prime factor q of n is found. A point $A(\neq \infty)$ is chosen such that $qA = \infty$. (In fact, n is not needed. Choose a point A' on E and find an integer n' with $n'A' = \infty$. There are ways of doing this, though it is not easy. Let q be a large prime factor of n', if it exists, and let $A = (n'/q)A'$. Then $qA = \infty$.) It is assumed that the message satisfies $0 \le m < q$. Alice chooses her secret integer a and computes $B = aA$. The public information is p, E, q, A, B. Alice does the following:

1. Chooses a random integer k with $1 \le k < q$

and computes R=kA=(x, y)

2. Computes $s \equiv k-1(m+ax) \pmod q$

3. Sends the signed message (m, R, s) to Bob

Bob verifies the signature as follows:

1. Computes $u1 \equiv s-1m \pmod q$ and $u2 \equiv s-1x \pmod q$

2. Computes V=u1A+u2B

3. Declares the signature valid if V=R

1. Show that the verification equation holds for a correctly signed message. Where is the fact that $qA=\infty$ used (see the "subtle point" mentioned in the ElGamal scheme in Section 21.5)?

2. Why does $k-1 \pmod q$ exist?

3. If q is large, why is there very little chance that s−1 does not exist mod q? How do we recognize the case when it doesn't exist? (Of course, in this case, Alice should start over by choosing a new k.)

4. How many computations "(large integer)×(point on E)" are made in the verification process here? How many are made in the verification process for the elliptic ElGamal scheme described in the text? (Compare with the end of Section 13.5.)

25. Let A and B be points on an elliptic curve and suppose B=kA for some integer k. Suppose also that $2nA=\infty$ for some integer n, but $T=2n-1A \neq \infty$.

    1. Show that if $k \equiv k' \pmod{2n}$, then $B=k'A$. Therefore, we may assume that $0 \leq k<2n$.

    2. Let j be an integer. Show that $jT=\infty$ when j is even and $jT \neq \infty$ when j is odd.

    3. Write $k=x0+2x1+4x2+\cdots+2n-1xn-1$, where each xi is 0 or 1 (binary expansion of k). Show that x0=0 if and only if $2n-1B=\infty$.

    4. Suppose that for some $m<n$ we know $x0, \ldots, xm-1$. Let $Qm=B-(x0+\cdots+2m-1xm-1)A$. Show that $2n-m-1Qm=\infty$ if and only if xm=0. This allows us to find xm. Continuing in this way, we obtain

$x_0, \ldots, x_{n-1}$, and therefore we can compute k. This technique can be extended to the case where $sA = \infty$, where s is an integer with only small prime factors. This is the analog of the Pohlig-Hellman algorithm (see Section 10.2).

21.7 Computer Problems

1. Let E be the elliptic curve $y^2 \equiv x^3+2x+3 \pmod{19}$.
    1. Find the sum $(1, 5)+(9, 3)$.
    2. Find the sum $(9, 3)+(9, -3)$.
    3. Using the result of part (b), find the difference $(1, 5)-(9, 3)$.
    4. Find an integer k such that $k(1, 5)=(9, 3)$.
    5. Show that $(1, 5)$ has exactly 20 distinct multiples, including ∞.
    6. Using (e) and Exercise 19(d), show that the number of points on E is a multiple of 20. Use Hasse's theorem to show that E has exactly 20 points.
2. You want to represent the message 12345 as a point (x, y) on the curve $y^2 \equiv x^3+7x+11 \pmod{593899}$. Write x=12345_ and find a value of the missing last digit of x such that there is a point on the curve with this x-coordinate.

3.

1. Factor 3900353 using elliptic curves.

2. Try to factor 3900353 using the $p-1$ method of Section 9.4. Using the knowledge of the prime factors obtained from part (a), explain why the $p-1$ method does not work well for this problem.

4. Let $P=(2, 3)$ be a point on the elliptic curve $y2 \equiv x3-10x+21 \pmod{557}$.

1. Show that $189P=\infty$, but $63P \neq \infty$ and $27P \neq \infty$.

2. Use Exercise 20 to show that P has order 189.

3. Use Exercise 19(d) and Hasse's theorem to show that the elliptic curve has 567 points.

5. Compute the difference $(5, 9)-(1, 1)$ on the elliptic curve $y2 \equiv x3-11x+11 \pmod{593899}$. Note that the answer involves large integers, even though the original points have small coordinates.

Chapter 22 Pairing-Based Cryptography

As we pointed out in the previous chapter, elliptic curves have various advantages in cryptosystems based on discrete logs. However, as we'll see in this chapter, they also open up exciting new vistas for cryptographic applications. The existence of a bilinear pairing on certain elliptic curves is what makes this possible.

First, we'll describe one of these pairings. Then we'll give various applications, including identity-based encryption, digital signatures, and encrypted keyword searches.

22.1 Bilinear Pairings

Although most of this chapter could be done in the context of cyclic groups of prime order, the primary examples of pairings in cryptography are based on elliptic curves or closely related situations. Therefore, for concreteness, we use only the following situation.

Let p be a prime of the form 6q−1, where q is also prime. Let E be the elliptic curve $y2 \equiv x3+1 \bmod p$. We need the following facts about E.

1. There are exactly p+1=6q points on E.

2. There is a point $P0 \neq \infty$ such that $qP0=\infty$. In fact, if we take a random point P, then, with very high probability, $6P \neq \infty$ and 6P is a multiple of P0.

3. There is a function $\tilde{e}$ that maps pairs of points $(aP0, bP0)$ to qth roots of unity for all integers a, b. It satisfies the **bilinearity property**

   $$\tilde{e}(aP0, bP0)=\tilde{e}(P0, P0)^{ab}$$

   for all a, b. This implies that

   $$\tilde{e}(aP, bQ)=\tilde{e}(P, Q)^{ab}$$

   for all points P, Q that are multiples of P0. (See Exercise 2.)

4. If we are given two points P and Q that are multiples of P0, then $\tilde{e}(P, Q)$ can be computed quickly from the coordinates of P and Q.

5. $\tilde{e}(P0, P0) \neq 1$, so it is a nontrivial qth root of unity.

We also make the following assumption:

1. If we are given a random point $P \neq \infty$ on E and a random qth root of unity $\omega \neq 1$, it is computationally infeasible to find a point Q on E with $\tilde{e}(Q, P)=\omega$ and it is computationally infeasible to find Q' with $\tilde{e}(P, Q')=\omega$.

Remarks

Properties (1) and (2) are fairly easy to verify (see Exercises 4 and 5). The existence of $\tilde{e}$ satisfying (3), (4), (5) is deep. In fact, $\tilde{e}$ is a modification of what is known as the Weil pairing in the theory of elliptic curves. The usual Weil pairing e satisfies $e(P0, P0)=1$, but the present version is modified using special properties of E to obtain (5). It is generally assumed that (A) is true if the prime p is large enough, but this is not known. See Exercise 10.

The fact that $\tilde{e}(P, Q)$ can be computed quickly needs some more explanation. The two points P, Q satisfy $P=aP0$ and $Q=bP0$ for some a, b. However, to find a and b requires solving a discrete log

problem, which could take a long time. Therefore, the obvious solution of choosing a random qth root of unity for e˜(P0, P0) and then using the bilinearity property to define e˜ does not work, since it cannot be computed quickly. Instead, e˜(P, Q) is computed directly in terms of the coordinates of the points P, Q.

Although we will not need to know this, the qth roots of unity lie in the finite field with p2 elements (see Section 3.11).

For more about the definition of e˜, see [Boneh-Franklin] or [Washington].

The curve E is an example of a **supersingular** elliptic curve, namely

one where the number of points is congruent to 1 mod p. (See Exercise 4.) For a while, these curves were regarded as desirable for cryptographic purposes because computations can be done quickly on them. But then it was shown that the discrete logarithm problem for them is only slightly more difficult than the classical discrete logarithm mod p (see Section 22.2), so they fell out of favor (after all, they are slower computationally than simple multiplication mod p, and they provide no security advantage). Because of the existence of the pairing e˜, they have become popular again.

22.2 The MOV Attack

Let E be the elliptic curve from Section 22.1. Suppose Q=kP0, where k is some integer and P0 is the point from Section 22.1. The Elliptic Curve Discrete Log Problem asks us to find k. Menezes, Okamoto, and Vanstone showed the following method of reducing this problem to the discrete log problem in the field with p2 elements. Observe that

e˜(Q, P0)=e˜(kP0, P0)=e˜(P0, P0)k.

Therefore, solving the discrete log problem e˜(Q, P0)=e˜(P0, P0)k for k yields the answer. Note that this latter discrete log problem is not on the elliptic curve, but instead in the finite

field with p2 elements. There are analogues of the index calculus for this situation, so usually this is an easier discrete log problem.

For a randomly chosen (not necessarily supersingular) elliptic curve, the method still works in theory. But the values of the pairing usually lie in a field much larger than the field with p2 elements. This slows down the computations enough that the MOV attack is infeasible for most non-supersingular curves.

The MOV attack shows that cryptosystems based on the elliptic curve discrete log problem for supersingular curves gives no substantial advantage over the classical

discrete log problem mod a prime. For this reason, supersingular curves were avoided for cryptographic purposes until these curves occurred in applications where pairings needed to be computed quickly, as in the next few sections.

22.3 Tripartite Diffie-Hellman

Alice, Bob, and Carlos want to agree on a common key (for a symmetric cryptosystem). All communications among them are public. If there were only two people, Diffie-Hellman could be used. A slight extension of this procedure works for three people:

1. Alice, Bob, and Carlos agree on a large prime p and a primitive root α.

2. Alice chooses a secret integer a, Bob chooses a secret integer b, and Carlos chooses a secret integer c.

3. Alice computes $A \equiv \alpha^a \pmod{p}$, Bob computes $B \equiv \alpha^b \pmod{p}$, and Carlos computes $C \equiv \alpha^c \pmod{p}$.

4. Alice sends A to Bob, Bob sends B to Carlos, and Carlos sends C to Alice.

5. Alice computes $A' \equiv Ca$, Bob computes $B' \equiv Ab$, and Carlos computes $C' \equiv Bc$.

6. Alice sends A' to Bob, Bob sends B' to Carlos, and Carlos sends C' to Alice.

7. Alice computes $A'' \equiv C'a$, Bob computes $B'' \equiv A'b$, and Carlos computes $C'' \equiv B'c$. Note that $A''=B''=C'' \equiv \alpha abc \pmod p$.

8. Alice, Bob, and Carlos use some agreed-upon method to obtain keys from A'', B'', C''. For example, they could use some standard hash function and apply it to $\alpha abc \pmod p$.

This protocol could also be used with p and α replaced by an elliptic curve E and a point P0, so Alice computes aP0, etc., and the final result is abcP0.

In 2000, Joux showed how to use pairings to obtain a more efficient protocol, one in which there is only one

round instead of two:

1. Alice, Bob, and Carlos choose a supersingular elliptic curve E and a point P0, as in Section 22.1.

2. Alice chooses a secret integer a, Bob chooses a secret integer b, and Carlos chooses a secret integer c.

3. Alice computes A=aP0, Bob computes B=bP0, and Carlos computes C=cP0.

4. Alice makes A public, Bob makes B public, and Carlos makes C public.

5. Alice computes e˜(B, C)a, Bob computes e˜(A, C)b, and Carlos computes e˜(A, B)c. Note that each person has computed e˜(P0, P0)abc.

6. Alice, Bob, and Carlos use some agreed-upon method to obtain keys from e˜(P0, P0)abc. For example, they could apply some standard hash function to this number.

The eavesdropper Eve sees E and the points P0, aP0, bP0, cP0 and needs to compute e˜(P0, P0)abc. This

computation is called the **Bilinear Diffie-Hellman Problem**. It is not known how difficult it is. However, if Eve can solve the Computational Diffie-Hellman Problem (see Section 10.4), then she uses E, P0, aP0, bP0 to obtain abP0 and computes e˜(abP0, cP0)=e˜(P0, P0)abc. Therefore, the Bilinear Diffie-Hellman Problem is no harder than the Computational Diffie-Hellman Problem.

Joux's result showed that pairings could be used in a constructive way in cryptography, rather than only in a destructive method such as the MOV attack, and this led to pairings being considered for applications such as those in the next few sections. It also

meant that supersingular curves again became useful in cryptography, with the added requirement that when a curve mod p is used, the prime p must be chosen large enough that the classical discrete logarithm problem (solve $\alpha x \equiv \beta \pmod{p}$ for x) is intractable.

22.4 Identity-Based Encryption

In most public key systems, when Alice wants to send a message to Bob, she looks up his public key in a directory and then encrypts her message. However, she needs some type of authentication – perhaps the directory has been modified by Eve, and the public key listed for Bob was actually created by Eve. Alice wants to avoid this situation. It was suggested by Shamir in 1984 that it would be nice to have an identity-based system, where Bob's public identification information (for example, his email address) serves as his public key. Such a system was

finally designed in 2001 by Boneh and Franklin.

Of course, some type of authentication of each user is still needed. In the present system, this occurs in the initial setup of the system during the communications between the Trusted Authority and the User. In the following, we give the basic idea of the system. For more details and improvements, see [Boneh-Franklin].

We need two public hash functions:

1. H1 maps arbitrary length binary strings to multiples of P0. A little care is needed in defining H1, since no one should be able, given a binary string b, to find k with H1(b)=kP0. See Exercise 7.

2. H2 maps qth roots of unity to binary strings of length n, where n is the length of the messages that will be sent. Since H2 must be specified before

the system is set up, this limits the lengths of the messages that can be sent. However, the message could be, for example, a DES key that is used to encrypt a much longer message, so this length requirement is not a severe restriction.

To set up the system we need a Trusted Authority. Let's call him Arthur. Arthur does the following.

1. He chooses, once and for all, a secret integer s. He computes P1=sP0, which is made public.

2. For each User, Arthur finds the user's identification *ID* (written as a binary string) and computes

   DUser=s H1(ID).

   Recall that H1(ID) is a point on E, so D User is s times this point.

3. Arthur uses a secure channel to send D User to the user, who keeps it secret. Arthur does not need to store D User, so he discards it.

The system is now ready to operate, but

first let's review what is known:

Public: E, p, P0, P1, H1, H2

Secret: s (known only to Arthur),
D User (one for each User; it is known only by that User)

Alice wants to send an email message m (of binary length n) to Bob, who is one of the Users. She knows Bob's address, which is `bob@computer.com`. This is his *ID*. Alice does the following.

1. She computes g=e˜(H1 (`bob@computer.com`), P1). This is a qth root of unity.

2. She chooses a random r ≡ 0 (mod q) and computes

   t=m⊕H2(gr).

3. She sends Bob the ciphertext

   c=(rP0, t).

Note that rP0 is a point on E, and t is a binary string of length n.

If Bob receives a pair (U, v), where U is a point on E and v is a binary string of length n, then he does the following.

1. He computes h=e˜(D Bob, U), which is a qth root of unity.

2. He recovers the message as

   m=v⊕H2(h).

Why does this yield the message? If the encryption is performed correctly, Bob receives U=rP0 and v=t=m⊕H2(gr). Since D Bob=s H1 (bob@computer.com),

h=e˜(D Bob, rP0)=e˜(H1, P0)sr=e˜(H1, sP0)r=gr.(22.1)

Therefore,

t⊕H2(h)=t⊕H2(gr)=m⊕H2(gr)⊕H2(gr)=m,

as desired. Note that the main step is Equation (22.1), which removes the secret s from the D Bob in the first argument of e˜ and puts it on the P0 in the second argument. This follows from the bilinearity property of the function e˜. Almost all of the cryptographic uses of pairings have a similar idea of moving from one masking of the secret to another. The pairing allows this to be done without knowing the value of s.

It is very important that s be kept secret. If Eve obtains s, then she can compute the points D User for each user and read every email. Since P1=sP0, the security of s is compromised if Eve can compute discrete logs on the elliptic curve. Moreover, the ciphertext contains rP0. If

Eve can compute a discrete log and find r, then she can compute gr and use this to find H2(gr) and also m. Therefore, for the security of the system, it is vital that p be chosen large enough that discrete logs are computationally infeasible.

22.5 Signatures

22.5.1 BLS Signatures

Alice wants to sign a document m. In earlier chapters, we have seen how to do this with RSA and ElGamal signatures. The BLS method, due to Boneh, Lynn, and Schacham, uses pairings.

We use a supersingular elliptic curve E0 and point P0, as in Section 22.1. To set up the signature scheme, we'll need a public hash function H that maps arbitrary length binary strings to multiples of P0. A little care is needed in defining H, since no one should be able, given a binary string b, to find k

with H(b)=kP0. See Exercise 7.

To set up the system, Alice chooses, once and for all, a secret integer a and computes KAlice=aP0, which is made public.

Alice's signature for the message m is S=aH(m), which is a point on E.

To verify that (m, S) is a valid signed message, Bob checks

e˜(S, P0)=?e˜(H(m), KAlice).

If this equation holds, Bob says that the signature is valid.

If Alice signs correctly,

e˜(S, P0)=e˜(aH(m), P0)=e˜(H(m), P0)a=e˜(H(m), aP0)=e˜(H(m), KAlice),

so the verification equation holds.

Suppose Eve wants to forge Alice's signature on a document m. The values of H(m), KAlice, and P0 are then already determined, so the verification equation says that Eve needs to find S satisfying e˜(S, P0)= a known quantity. Assumption (A) is Section 22.1 says that (we hope) this is computationally infeasible.

22.5.2 A Variation

The BLS signature scheme uses a hash function whose values are points on an elliptic curve. This might seem less natural than using a standard hash function with values that are binary strings (that is, numbers). The following

method of Zhang, Safavi-Naini, and Susilo remedies this situation. Let H be a standard hash function such as SHA-3 or SHA-256 that maps binary strings of arbitrary length to binary strings of fixed length. Regard the output of H as an integer. Alice's key is the same as in BLS, namely, KAlice=aP0. But the signature is computed as

$$S=(H(m)+a)-1P0,$$

where (H(m)+a)−1 is the modular multiplicative inverse of (H(m)+a) mod q (where q is the order of P0, as in Section 22.1).

The verification equation for the signed message (m, S) is

$$e\tilde{}(H(m)P0+KAlice, S)=?e\tilde{}(P0, P0).$$

Since H(m)P0+KAlice=(H(m)+a)P0, the left side of the verification equation equals the right side when Alice signs correctly. Again, assumption (A) from Section 22.1 says that it should be hard for Eve to forge Alice's signature.

22.5.3 Identity-Based Signatures

When Alice uses one of the above methods or uses RSA or ElGamal signatures to sign a document, and Bob wants to verify the signature, he looks up Alice's key on a web-page, for example, and uses it in the verification process. This means he must trust the web-page to be correct, not a fake one made by Eve. It would be preferable to use something closely associated with

Alice such as her email address as the public key. This is of course the same problem that was solved in the previous section for encyprytion, and similar techniques work in the present situation.

The following method of Hess gives an identity-based signature scheme.

We use a supersingular elliptic curve E and point P0 as in Section 22.1. We need two public hash functions:

1. H1 maps arbitrary length binary strings to multiples of P0. A little care is needed in defining H1, since no one should be able, given a binary string b, to find k with H1(b)=kP0. See Exercise 7.

2. H2 maps binary strings of arbitrary length to binary strings of fixed length; for example, H2 can be a standard hash function such as SHA-3 or SHA-256.

To set up the system, we need a Trusted Authority. Let's call him Arthur. Arthur does the following.

1. He chooses, once and for all, a secret integer s. He computes P1=sP0, which is made public.

2. For each User, Arthur finds the user's identification *ID* (written as a binary string) and computes

   DUser=s H1(ID).

   Recall that H1(ID) is a point on E, so D User is s times this point.

3. Arthur uses a secure channel to send D User to the user, who keeps it secret. Arthur does not need to store D User, so he discards it.

The system is now ready to operate, but first let's review what is known:

Public: E, p, P0, P1, H1, H2

Secret: s (known only to Arthur),

D User (one for each User; it is known only by that User)

To sign m, Alice does the following.

1. She chooses a random point $P \neq \infty$ on E.
2. She computes r=e˜(P, P0).
3. She computes h=H2(m‖r).
4. She computes U=hDAlice+P.
5. The signed message is (m, U, h).

If Bob receives a triple (m, U, h), where U is a point on E and h is a binary string, then he does the following.

1. He computes v1=e˜(H1(IDAlice), P1)−h, which is a qth root of unity.
2. He computes v2=v1·e˜(U, P0).
3. If h=H2(m‖v2), then Bob says the signature is valid.

Let's suppose Alice signed correctly. Then, writing H1 for H1(IDAlice), we have

$$v1=e\tilde{}(H1, P1)-h=e\tilde{}(H1, P0)-hs=e\tilde{}(sH1, P0)-h=e\tilde{}(DAlice, P0)-h=e\tilde{}(-h\,DAlice, P0).$$

Also,

$$v2=v1\cdot e\tilde{}(U, P0)=e\tilde{}(-h\,DAlice, P0)\cdot e\tilde{}(U, P0)=e\tilde{}(-h\,DAlice+U, P0)=e\tilde{}(P, P0)=r.$$

Therefore,

$$H2(m\|v2)=H2(m\|r)=h,$$

so the signature is declared valid.

Suppose Eve has a document m and she wants to forge Alice's signature so that (m, U, h) is a valid signature. She cannot choose h arbitrarily since Bob is going to compute v2 and see whether h=H2(m||v2). Therefore, if H2 is a good

hash function, Eve's best strategy is to choose a value v2′ and compute h=H(m‖ v2′). Since H2 is collision resistant and H2(m‖v2′)=h=H2(m‖v2), this v2′ should equal v2=v1·e˜(U, P0). But v1 is completely determined by Alice's ID and h. This means that in order to satisfy the verification equation, Eve must find U such that e˜(U, P0) equals a given quantity. Assumption (A) says this should be hard to do.

22.6 Keyword Search

Alice runs a large corporation. Various employees send in reports containing secret information. These reports need to be sorted and routed to various departments, depending on the subject of the message. Of course, each report could have a set of keywords attached, and the keywords could determine which departments receive the reports. But maybe the keywords are sensitive information, too. Therefore, the keywords need to be encrypted. But if the person who sorts the reports decrypts the keywords, then this person sees the sensitive keywords. It would be good to have the keywords encrypted in

a way that an authorized person can search for a certain keyword in a message and determine either that the keyword is present or not, and receive no other information about other keywords. A solution to this problem was found by Boneh, Di Crescenzo, Ostrovsky, and Persiano.

Start with a supersingular elliptic curve E and point P0 as in Section 22.1. We need two public hash functions:

1. H1 maps arbitrary length binary strings to multiples of P0. A little care is needed in defining H1, since no one should be able, given a binary string b, to find k with H1(b)=kP0. See Exercise 7.

2. H2 maps the qth roots of unity (in the finite field with p2 elements) to binary strings of fixed length; for example, if the roots of unity are expressed in binary, H2 can be a standard hash function.

Alice sets up the system as follows. Incoming reports (encrypted) will have attachments consisting of encrypted keywords. Each department will have a set of keyword searches that it is allowed to do. If it finds one of its allotted keywords, then it saves the report.

1. Alice chooses a secret integer a and computes $P1=aP0$.

2. Alice computes

   $Tw=a\,H1(w)$.

   The point Tw is sent via secure channels to each department that is authorized to search for w.

When Daphne writes a report, she attaches the relevant encrypted keywords to the documents. These encrypted keywords are produced as

follows:

1. Let w be a keyword. Daphne chooses a random integer $r \equiv 0 \pmod{q}$ and computes

   t=e˜(H1(w), rP1).

2. The encryption of the keyword w is the pair

   [rP0, H2(t)].

A searcher looks at the encrypted keywords [A, b] attached to a document and checks

H2(e˜(Tw, A))=?b.

If yes, then the searcher concludes that w is a keyword for that document. If no, then w is not a keyword for it.

Why does this work? If [A, b] is the encrypted form of the keyword w, then

A=rP0 and t=e˜(H1(w), rP1)

for some r. Therefore,

$$\tilde{e}(Tw, A)=\tilde{e}(a\,H1(w), rP0)=\tilde{e}(H1(w), rP0)a=\tilde{e}(H1(w), rP1)=t,$$

so $H2(\tilde{e}(Tw, A))=H2(t)=b$.

Suppose conversely, that w' is another keyword. Is it possible that $[A, b]$ corresponds to both w and w'? Since the same value of r could be used in the encryptions of w and w', it is possible that A occurs in encryptions of both w and w'. However, we'll show that in this case the number b comes from at most one of w and w'. Since $H1$ is collision resistant and $Tw'=a\,H1(w')$ and $Tw=a\,H1(w)$, we expect that $Tw' \neq Tw$ and $\tilde{e}(Tw', A) \neq \tilde{e}(Tw, A)$. (See Exercise 1.) Since $H2$ is collision resistant, we expect that

H2(e˜(Tw′, A)) ≠ H2(e˜(Tw, A)).

Therefore, [A, b] passes the verification equation for at most one keyword w.

Each time that Daphne encrypts the keyword, a different r should be used. Otherwise, the encryption of the keyword will be the same and this can lead to information being leaked. For example, someone could notice that certain keywords occur frequently and make some guesses as to their meanings.

There are many potential applications of this keyword search scheme. For example, suppose a medical researcher wants to find out how many patients at a certain hospital were treated for disease X in the previous year. For privacy

reasons, the administration does not want the researcher to obtain any other information about the patients, for example, gender, race, age, and other diseases. The administration could give the researcher TX. Then the researcher could search the encrypted medical records for keyword X without obtaining any information other than the presence or absence of X.

22.7 Exercises

1. Let E be the supersingular elliptic curve from Section 22.1.

    1. Let $P \neq \infty$ and $Q \neq \infty$ be multiples of P0. Show that $\tilde{e}(P, Q) \neq 1$. (Hint: Use the fact that $\tilde{e}(P0, P0)=\omega$ is a qth root of unity and that $\omega x=1$ if and only if $x \equiv 0 \pmod{q}$.)

    2. Let $Q \neq \infty$ be a multiple of P0 and let P1, P2 be multiples of P0. Show that if $\tilde{e}(P1, Q)=\tilde{e}(P2, Q)$, then $P1=P2$.

2. Let E be the supersingular elliptic curve from Section 22.1.

    1. Show that

        $$\tilde{e}(aP, bQ)=\tilde{e}(P, Q)^{ab}$$

        for all points P, Q that are multiples of P0.

    2. Show that

$$\tilde{e}(P+Q, R)=\tilde{e}(P, R)\,\tilde{e}(Q, R)$$

for all P, Q, R that are multiples of P0.

3. Let E be the supersingular elliptic curve from Section 22.1. Suppose you have points A, B on E that are multiples of P0 and are not equal to ∞. Let a and b be two secret integers. Suppose you are given the points aA and bB. Find a way to use $\tilde{e}$ to decide whether or not $a \equiv b \pmod{q}$.

4. Let $p \equiv -1 \pmod{3}$ be prime.

    1. Show that there exists d with $3d \equiv 1 \pmod{p-1}$.

    2. Show that if $a3 \equiv b \pmod{p}$ if and only if $a \equiv bd \pmod{p}$. This shows that every integer mod p has a unique cube root.

    3. Show that $y2 \equiv x3+1 \pmod{p}$ has exactly p+1 points (including the point ∞). (Hint: Apply part (b) to $y2-1$.) (Remark: A curve mod p whose number of points is congruent to 1 mod p is called *supersingular*.)

5. (for those who know some group theory)

    1. In the situation of Exercise 4, suppose

that $p=6q-1$ with q also prime. Show that there exists a point $P0 \neq \infty$ such that $qP0=\infty$.

2. Let $Q=(2, 3)$, as in Exercise 9 in Chapter 21. Show that if $P \in \{\infty, Q, 2Q, 3Q, 4Q, 5Q\}$, then $6P \neq \infty$ and $6P$ is a multiple of $P0$. (For simplicity, assume that $q>3$.)

6. Let $H0$ be a hash function that takes a binary string of arbitrary length as input and then outputs an integer mod p. Let $p=6q-1$ be prime with q also prime. Show how to use $H0$ to construct a hash function $H1$ that takes a binary string of arbitrary length as input and outputs a point on the elliptic curve $y2 \equiv x3+1 \pmod{p}$ that is a multiple of the point $P0$ of Section 22.1. (Hint: Use the technique of Exercise 4 to find y, then x. Then use Exercise 5(b).)

7.

1. In the identity-based encryption system of Section 22.4, suppose Eve can compute k such that $H1$ `(bob@computer.edu)` $=k\,P0$.. Show that Eve can compute gr and therefore read Bob's messages.

2. In the BLS signature scheme of Section

22.5.1, suppose Eve can compute k such that H(m)=k P0. Show that Eve can compute S such that (m, S) is a valid signed document.

3. In the identity-based signature scheme of Section 22.5.1, suppose Eve can compute k such that H1(IDAlice)=k P0. Show that Eve can compute DAlice and therefore forge Alice's signature on documents.

4. In the keyword search scheme of Section 22.6, suppose Eve can compute k such that H1(w)=k P0. Show that Eve can compute Tw and therefore find the occurrences of encrypted w on documents.

8. Let E and P0 be as in Section 22.1. Show that an analogue of the Decision Diffie-Hellman problem can be solved for E. Namely, if we are given aP0, bP0, cP0, show how we can decide whether abP0=cP0.

9. Suppose you try to set up an identity-based cryptosystem as follows. Arthur chooses large primes p and q and forms n=pq, which is made public. For each User, he converts the User's identification ID to a number e User by some

public method and then computes d with de User $\equiv 1 \pmod{\phi(n)}$. Arthur gives d to the User. The integer n is the same for all users. When Alice wants to send an email to Bob, she uses the public method to convert his email address to e Bob and then uses this to encrypt messages with RSA. Bob knows d, so he can decrypt. Explain why this system is not secure.

10. You are given a point $P \neq \infty$ on the curve E of Section 22.1 and you are given a qth root of unity ω. Suppose you can solve discrete log problems for the qth roots of unity. That is, if $\alpha \neq 1$ and β are qth roots of unity, you can find k so that $\alpha k=\beta$. Show how to find a point Q on E with $e\tilde{}(Q, P)=\omega$.

Chapter 23 Lattice Methods

Lattices have become an important tool for the cryptanalyst. In this chapter, we give a sampling of some of the techniques. In particular, we use lattice reduction techniques to attack RSA in certain cases. Also, we describe the NTRU public key system and show how it relates to lattices. For a more detailed survey of cryptographic applications of lattices, see [Nguyen-Stern].

23.1 Lattices

Let v1, …, vn be linearly independent vectors in n-dimensional real space Rn. This means that every n-dimensional real vector v can be written in the form

$$v=a_1v_1+\cdots+a_nv_n$$

with real numbers a1, …, an that are uniquely determined by v. The **lattice** generated by v1, …, vn is the set of vectors of the form

$$m_1v_1+\cdots+m_nv_n$$

where m1, …, mn are integers. The set {v1, …, vn} is called a **basis** of the lattice. A lattice has infinitely many possible bases. For example, suppose

{v1, v2} is a basis of a lattice. Let k be an integer and let w1=v1+kv2 and w2=v2. Then {w1, w2} is also a basis of the lattice: Any vector of the form m1v1+m2v2 can be written as m1′w1+m2′w2 with m1′=m1 and m2′=m2−km1, and similarly any integer linear combination of w1 and w2 can be written as an integer linear combination of v1 and v2.

Example

Let v1=(1, 0) and v2=(0, 1). The lattice generated by v1 and v2 is the set of all pairs (x, y) with x, y integers. Another basis for this lattice is {(1, 5), (0, 1)}. A third basis is {(5, 16), (6, 19)}. More generally, if abcd is a matrix with determinant ±1, then {(a, b), (c, d)} is a

basis of this lattice (Exercise 4).

The length of a vector v=(x1, …, xn) is

‖v‖=x12+⋯+xn21/2.

Many problems can be related to finding a shortest nonzero vector in a lattice. In general, the **shortest vector problem** is hard to solve, especially when the dimension of the lattice is large. In the following section, we give some methods that work well in small dimensions.

Example

A shortest vector in the lattice generated by

(31, 59) and (37, 70)

is $(3, -1)$ (another shortest vector is $(-3, 1)$). How do we find this vector? This is the subject of the next section. For the moment, we verify that $(3, -1)$ is in the lattice by writing

$$(3, -1) = -19(31, 59) + 16(37, 70).$$

In fact, $\{(3, -1), (1, 4)\}$ is a basis of the lattice. For most purposes, this latter basis is much easier to work with than the original basis since the two vectors $(3, -1)$ and $(1, 4)$ are almost orthogonal (their dot product is $(3, -1) \cdot (1, 4) = -1$, which is small). In contrast, the two vectors of the original basis are nearly parallel and have a very large dot product. The methods of the next section show how to replace a basis of a lattice with a new basis whose vectors

are almost orthogonal.

23.2 Lattice Reduction

23.2.1 Two-Dimensional Lattices

Let v1, v2 form the basis of a two-dimensional lattice. Our first goal is to replace this basis with what will be called a reduced basis.

If $\|v1\| > \|v2\|$, then swap v1 and v2, so we may assume that $\|v1\| \leq \|v2\|$. Ideally, we would like to replace v2 with a vector v2* perpendicular to v1. As in the Gram-Schmidt process from linear algebra, the vector

$$v2^* = v2 - \frac{v1 \cdot v2}{v1 \cdot v1} v1 \quad (23.1)$$

is perpendicular to v1. But this vector might not lie in the lattice. Instead, let t be the closest integer to (v1·v2)/(v1·v1) (for definiteness, take 0 to be the closest integer to ±12, and ±1 to be closest to ±32, etc.). Then we replace the basis {v1, v2} with the basis

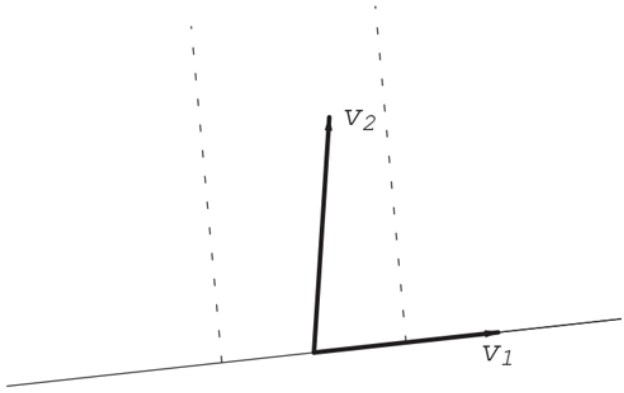
{v1, v2−tv1}.

We then repeat the process with this new basis.

We say that the basis {v1, v2} is **reduced** if

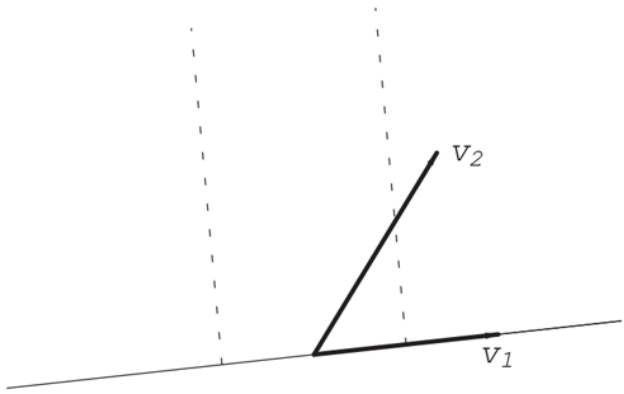
‖v1‖ ≤ ‖v2‖ and −12 ≤ v1·v2v1·v1 ≤ 12.

The above reduction process stops exactly when we obtain a reduced basis, since this means that t=0.



A reduced basis

23.2-1 Full Alternative Text



A nonreduced basis

23.2-2 Full Alternative Text

In the figures, the first basis is reduced because v2 is longer than v1 and the projection of v2 onto v1 is less than half the length of v1. The second basis is nonreduced because the projection of v2

onto v1 is too long. It is easy to see that a basis {v1, v2} is reduced when v2 is at least as long as v1 and v2 lies within the dotted lines of the figures.

Example

Let's start with v1=(31, 59) and v2=(37, 70). We have ‖v1‖<‖v2‖, so we do not swap the two vectors. Since

$$v1 \cdot v2 v1 \cdot v1 = 52774442,$$

we take t=1. The new basis is

$$v1' = v1 = (31, 59) \text{ and } v2' = v2 - v1 = (6, 11).$$

Swap v1′ and v2′ and rename the vectors to obtain a basis

$$v1'' = (6, 11) \text{ and } v2'' = (31, 59).$$

We have

v1″·v2″v1″·v1″=835157,

so we take t=5. This yields vectors

(6, 11) and(1, 4)=(31, 59)−5·(6, 11).

Swap these and name them v1(3)=(1, 4) and v2(3)=(6, 11). We have

v1(3)·v2(3)v1(3)·v1(3)=5017,

so t=3. This yields, after a swap,

v1r=(3, −1)andv2r=(1, 4).

Since ||v1r|| ≤ ||v2r|| and

v1r·v2rv1r·v1r=−110,

the basis {v1r, v2r} is reduced.

A natural question is whether this process always produces a reduced basis. The answer is yes, as we prove in the following theorem. Moreover, the

first vector in the reduced basis is a shortest vector for the lattice.

We summarize the discussion in the following.

Theorem

Let {v1, v2} be a basis for a two-dimensional lattice in R2. Perform the following algorithm:

1. If $\|v1\|>\|v2\|$, swap v1 and v2 so that $\|v1\| \leq \|v2\|$.
2. Let t be the closest integer to $(v1 \cdot v2)/(v1 \cdot v1)$.
3. If $t=0$, stop. If $t \neq 0$, replace v2 with $v2-tv1$ and return to step 1.

The algorithm stops in finite time and yields a reduced basis {v1r, v2r} of the lattice. The vector v1r is a shortest

nonzero vector for the lattice.

Proof

First we prove that the algorithm eventually stops. As in Equation 23.1, let $\mu=(v1\cdot v2)/(v1\cdot v1)$ and let $v2*=v2-\mu v1$. Then

$$v2-tv1=v2*+(\mu-t)v1.$$

Since v1 and v2* are orthogonal, the Pythagorean theorem yields

$$\|v2-tv1\|2=\|v2*\|2+\|(\mu-t)v1\|2=\|v2*\|2+(\mu-t)2\|v1\|2.$$

Also, since $v2=v2*+\mu v1$, and again since v1 and v2* are orthogonal,

$$\|v2\|2=\|v2*\|2+\mu2\|v1\|2.$$

Note that if $-1/2 \le \mu \le 1/2$ then $t=0$ and $\mu-t=\mu$. Otherwise, $|\mu-t| \le 1/2t|\mu|$.

Therefore, if $t \neq 0$, we have $|\mu - t| < |\mu|$, which implies that

$$\|v2 - tv1\|2 = \|v2^*\|2 + (\mu - t)2\|v1\|2 < \|v2^*\|2 + \mu 2\|v1\|2 = \|v2\|2.$$

Therefore, if the process continues forever without yielding a reduced basis, then the lengths of the vectors decrease indefinitely. However, there are only finitely many vectors in the lattice that are shorter than the original basis vector v2. Therefore, the lengths cannot decrease forever, and a reduced basis must be found eventually.

To prove that the vector v1 in a reduced basis is a shortest nonzero vector for the lattice, let $av1+bv2$ be any nonzero vector in the lattice, where a and b are integers. Then

$$\|av1+bv2\|2=(av1+bv2)\cdot(av1+bv2)=a2\|v1\|2+b2\|v2\|2+2abv1\cdot v2.$$

Because {v1, v2} is reduced,

$$-12v1\cdot v1 \le v1\cdot v2 \le 12v1\cdot v1,$$

which implies that $2abv1\cdot v2 \ge -|ab|\|v1\|2$. Therefore,

$$\| av1+bv2 \|2=(av1+bv2)\cdot(av1+bv2)=a2\| v1 \|2+2abv1\cdot v2+b2\| v2 \|2 \ge a2\| v1 \|2-| ab |\| v1 \|2+b2\| v2 \|2 \ge a2\| v1 \|2-| ab |\| v1 \|2+b2\| v1 \|2,$$

since $\|v2\|2 \ge \|v1\|2$ by assumption. Therefore,

$$\|av1+bv2\|2 \ge (a2-|ab|+b2)\|v1\|2.$$

But $a2-|ab|+b2$ is an integer. Writing it as $(|a|-12|b|)2+14|b|2$, we see that it is nonnegative, and it equals 0 if and only if $a=b=0$. Since $av1+bv2 \ne 0$, we must have $a2-|ab|+b2 \ge 1$. Therefore,

$$\|av1+bv2\|2 \ge \|v1\|2,$$

so v1 is a shortest nonzero vector.

23.2.2 The LLL algorithm

Lattice reduction in dimensions higher than two is much more difficult. One of the most successful algorithms was invented by A. Lenstra, H. Lenstra, and L. Lovász and is called the *LLL* algorithm. In many problems, a short vector is needed, and it is not necessary that the vector be the shortest. The *LLL* algorithm takes this approach and looks for short vectors that are almost as short as possible. This modified approach makes the algorithm run very quickly (in what is known as polynomial time). The algorithm performs calculations similar to those in the two-dimensional case, but the steps are more technical, so

we omit details, which can be found in [Cohen], for example. The result is the following.

Theorem

Let L be the n-dimensional lattice generated by v1, …, vn in Rn. Define the determinant of the lattice to be

$$D=|\det(v 1 , ..., v n)|.$$

(This can be shown to be independent of the choice of basis. It is the volume of the parallelepiped spanned by v1, …, vn.) Let λ be the length of a shortest nonzero vector in L. The *LLL* algorithm produces a basis {b1, …, bn} of L satisfying

1. $\|b1\| \le 2(n-1)/4D1/n$

2. $\|b1\| \leq 2^{(n-1)/2}\lambda$

3. $\|b1\|\|b2\|\cdots\|bn\| \leq 2^{n(n-1)/4}D$.

Statement (2) says that b1 is close to being a shortest vector, at least when the dimension n is small. Statement (3) says that the new basis vectors are in some sense close to being orthogonal. More precisely, if the vectors b1, …, bn are orthogonal, then the volume D equals the product $\|b1\|\|b2\|\cdots\|bn\|$. The fact that this product is no more than $2^{n(n-1)/4}$ times D says that the vectors are mostly close to orthogonal.

The running time of the *LLL* algorithm is less than a constant times $n^6 \log^3 B$, where n is the dimension and B is a bound on the lengths of the original basis vectors. In practice it is much

faster than this bound. This estimate shows that the running time is quite good with respect to the size of the vectors, but potentially not efficient when the dimension gets large.

Example

Let's consider the lattice generated by (31, 59) and (37, 70), which we considered earlier when looking at the two-dimensional algorithm. The *LLL* algorithm yields the same result, namely b1=(3, −1) and b2=(1, 4). We have D=13 and λ=10 (given by ‖(3, −1)‖, for example). The statements of the theorem are

1. ‖b1‖=10 ≤ 21/413
2. ‖b1‖=10 ≤ 21/210

3. ‖b1‖‖b2‖=1017 ≤ 21/213.

23.3 An Attack on RSA

Alice wants to send Bob a message of the form

The answer is **

or

The password for your new account is ********.

In these cases, the message is of the form

m=B+x, where B is fixed and |x| ≤ Y

for some integer Y. We'll present an attack that works when the encryption exponent is small.

Suppose Bob has public RSA key

(n, e)=(n, 3). Then the ciphertext is

$$c \equiv (B+x)3 \pmod{n}.$$

We assume that Eve knows B, Y, and n, so she only needs to find x. She forms the polynomial

$$f(T)=(B+T)3-c=T3+3BT2+3B2T+B3-c \equiv T3+a2T2+a1T+a0 \pmod{n}.$$

Eve is looking for $|x| \leq Y$ such that $f(x) \equiv 0 \pmod{n}$. In other words, she is looking for a small solution to a polynomial congruence $f(T) \equiv 0 \pmod{n}$.

Eve applies the *LLL* algorithm to the lattice generated by the vectors

$$v1=(n, 0, 0, 0), \quad v2=(0, Yn, 0, 0),$$
$$v3=(0, 0, Y2n, 0), \; v4=(a0, a1Y, a2Y2, Y3).$$

This yields a new basis b1, …, b4, but we need only b1. The theorem in

Subsection 23.2.2 tells us that

‖b1‖ ≤ 23/4det(v1, …, v4)1/4(23.2)
=23/4(n3Y6)1/4=23/4n3/4Y3/2.(23.3)

We can write

b1=c1v1+⋯+c4v4=(e0, Ye1, Y2e2, Y3e3)

with integers ci and with

e0=c1n+c4a0e1=c2n+c4a1e2=c3n+c4a2e3=c4.

It is easy to see that

ei ≡ c4ai (mod n), 0 ≤ i ≤ 2.

Form the polynomial

g(T)=e3T3+e2T2+e1T+e0.

Then, since the integer x satisfies f(x) ≡ 0 (mod n) and since the coefficients of c4f(T) and g(T) are congruent mod n,

$$0 \equiv c4f(x) \equiv g(x) \pmod{n}.$$

Assume now that

$$Y<2-7/6n1/6.(23.4)$$

Then

$$|g(x)| \le |e0|+|e1x|+|e2x2|+|e3x3| \le |e0|+|e1|Y+|e2|Y2+|e3|Y3=(1, 1, 1, 1)\cdot|e0|, |e1Y|, |e2Y2|, |e3Y3| \le \|(1, 1, 1, 1)\|\|e0|, \dots, |e3Y3| =2\|b1\|,$$

where the last inequality used the Cauchy-Schwarz inequality for dot products (that is, $v \cdot w \le \|v\|\|w\|$). Since, by (17.3) and (17.4),

$$\|b1\| \le 23/4n3/4Y3/2<23/4n3/42-7/6n1/63/2=2-1n,$$

we obtain

$$|g(x)|<n.$$

Since $g(x) \equiv 0 \pmod{n}$, we must have $g(x)=0$. The zeros of $g(T)$ may be determined numerically, and we obtain

at most three candidates for x. Each of these may be tried to see if it gives the correct ciphertext. Therefore, Eve can find x.

Note that the above method replaces the problem of finding a solution to the congruence $f(T) \equiv 0 \pmod{n}$ with the exact, non-congruence, equation $g(T)=0$. Solving a congruence often requires factoring n, but solving exact equations can be done by numerical procedures such as Newton's method.

In exactly the same way, we can find small solutions (if they exist) to a polynomial congruence of degree d, using a lattice of dimension d+1. Of course, d must be small enough that *LLL* will run in a reasonable time.

Improvements to this method exist. Coppersmith ([Coppersmith2]) gave an algorithm using higher-dimensional lattices that looks for small solutions x to a monic (that is, the highest-degree coefficient equals 1) polynomial equation $f(T) \equiv 0 \pmod{n}$ of degree d. If $|x| \leq n^{1/d}$, then the algorithm runs in time polynomial in logn and d.

Example

Let

n=19278410554286974871575942589 17

(which happens to be the product of the primes p=757285757575769 and q=2545724696579693, but Eve does not know this). Alice is sending the message

The answer is **,

where ** denotes a two-digit number. Therefore the message is m=B+x where B=2008050001141923051800091900000 and 0 ≤ x<100. Suppose Alice sends the ciphertext c ≡ (B +x)3 ≡ 30326308498619648559464058932 (mod n). Eve forms the polynomial

f(T)=(B+T)3−c ≡ T3+a2T2+a1T+a0(mod n),

where

a2=602415000342576915540027570000a1=112354912400424746936217146

Note that a0 ≡ B3−c (mod n).

Eve uses *LLL* to find a root of f(T) (mod n). She lets Y=100 and forms the vectors

v1=(n, 0, 0, 0),v2=(0, 100n, 0, 0),v3=(0, 0, 104n, 0) v4=(a0, 100a1, 104a2, 106).

The *LLL* algorithm produces the vector

308331465484476402v1+589837092377839611v2+316253828707108264v3−

Eve then looks at the polynomial

g(T)=−1012071602751202635T3+4058485655851944008 8T2
−5778160874535342323853T+2460734306658871861084 74.

The roots of g(T) are computed numerically to be

42.000000000, −0.949612039±76.079608511i.

It is easily checked that g(42)=0, so the plaintext is

The answer is 42.

Of course, a brute force search through all possibilities for the two-digit number x could have been used to find the answer in this case. However, if n is taken to be a 200-digit number, then Y can have around 33 digits. A brute force

search would usually not succeed in this situation.

23.4 NTRU

If the dimension n is large, say $n \geq 100$, the *LLL* algorithm is not effective in finding short vectors. This allows lattices to be used in cryptographic constructions. Several cryptosystems based on lattices have been proposed. One of the most successful current systems is NTRU (rumored to stand for either "Number Theorists aRe Us" or "Number Theorists aRe Useful"). It is a public key system. In the following, we describe the algorithm for transmitting messages using a public key. There is also a related signature scheme, which we won't discuss. Although the initial description of NTRU does not involve

lattices, we'll see later that it also has a lattice interpretation.

First, we need some preliminaries. Choose an integer N. We will work with the set of polynomials of degree less than N. Let

$$f=a_{N-1}X^{N-1}+\cdots+a_0 \text{ and } g=b_{N-1}X^{N-1}+\cdots+b_0$$

be two such polynomials. Define

$$h=f*g=c_{N-1}X^{N-1}+\cdots+c_0,$$

where

$$c_i=\sum_{j+k\equiv i} a_j b_k.$$

The summation is over all pairs j, k with $j+k \equiv i \pmod{N}$.

For example, let N=3, let $f=X^2+7X+9$, and let $g=3X^2+2X+5$. Then the

coefficient c1 of X in f∗g is

$$a0b1+a1b0+a2b2=9\cdot2+7\cdot5+1\cdot3=56,$$

and

$$f*g=46X2+56X+68.$$

From a slightly more advanced viewpoint, f∗g is simply multiplication of polynomials mod XN−1 (see Exercise 5 and Section 3.11).

NTRU works with certain sets of polynomials with small coefficients, so it is convenient to have a notation for them. Let

L(j, k)=the set of polynomials of degree <Nwith j coefficients equal to +1and k coefficients equal to −1.The remaining coefficients are 0.

We can now describe the NTRU algorithm. Alice wants to send a

message to Bob, so Bob needs to set up his public key. He chooses three integers N, p, q with the requirements that gcd(p, q)=1 and that p is much smaller than q. Recommended choices are

(N, p, q)=(107, 3, 64)

for moderate security and

(N, p, q)=(503, 3, 256)

for very high security. Of course, these parameters will need to be adjusted as attacks improve. Bob then chooses two secret polynomials f and g with small coefficients (we'll say more about how to choose them later). Moreover, f should be invertible mod p and mod q, which means that there exist polynomials Fp and Fq of degree less

than N such that

$Fp*f \equiv 1 \pmod{p}$, $Fq*f \equiv 1 \pmod{q}$.

Bob calculates

$h \equiv Fq*g \pmod{q}$.

Bob's public key is

(N, p, q, h).

His private key is f. Although Fp can be calculated easily from f, he should store (secretly) Fp since he will need it in the decryption process. What about g? Since $g \equiv f*h \pmod{q}$, he is not losing information by not storing it (and he does not need it in decryption).

Alice can now send her message. She represents the message, by some prearranged procedure, as a polynomial

m of degree less than N with coefficients of absolute value at most (p−1)/2. When p=3, this means that m has coefficients −1, 0, 1. Alice then chooses a small polynomial ϕ (“small” will be made more precise shortly) and computes

$$c \equiv p\phi * h + m \pmod{q}.$$

She sends the ciphertext c to Bob.

Bob decrypts by first computing

$$a \equiv f * c \pmod{q}$$

with all coefficients of the polynomial a of absolute value at most q/2, then (usually) recovering the message as

$$m \equiv F_p * a \pmod{p}.$$

Why should this work? In fact, sometimes it doesn’t, but experiments

with the parameter choices given below indicate that the probability of decryption errors is less than 5×10−5. But here is why the decryption is usually correct. We have

$$a \equiv f*c \equiv f*(p\phi*h+m) \equiv f*p\phi*Fq*g+f*m \equiv p\phi*g+f*m \pmod{q}.$$

Since ϕ, g, f, m have small coefficients and p is much smaller than q, it is very probable that pϕ∗g+f∗m, before reducing mod q, has coefficients of absolute value less than q/2. In this case, we have equality

$$a=p\phi*g+f*m.$$

Then

$$Fp*a=pFp*\phi*g+Fp*f*m \equiv 0+1*m \equiv m \pmod{p},$$

so the decryption works.

For (N, p, q)=(107, 3, 64), the recommended choices for f, g, ϕ are

$f \in L(15, 14), g \in L(12, 12), \phi \in L(5, 5)$

(recall that this means that the coefficients of f are fifteen 1s, fourteen −1s, and the remaining 78 coefficients are 0).

For (N, p, q)=(503, 3, 256), the recommended choices for f, g, ϕ are

$f \in L(216, 215), g \in L(72, 72), \phi \in L(55, 55).$

With these choices of parameters, the polynomials f, g, ϕ are small enough that the decryption works with very high probability.

The reason f has a different number of 1s and −1s is so that $f(1) \neq 0$. It can be

shown that if f(1)=0, then f cannot be invertible.

Example

Let (N, p, q)=(5, 3, 16) (this choice of N is much too small for any security; we use it only in order to give an explicit example). Take f=X4+X−1 and g=X3−X. Since

(X3+X2−1)∗(X4+X−1) ≡ 1 (mod 3),

we have

Fp=X3+X2−1.

Also,

Fq=X3+X2−1h=−X4−2X3+2X+1 ≡ Fq∗g (mod 16).

Bob’s public key is

(N, p, q, h)=(5, 3, 16, −X4−2X3+2X+1).

His private key is

$$f = X^4 + X - 1.$$

Alice takes her message to be $m = X^2 - X + 1$. She chooses $\phi = X - 1$. Then the ciphertext is

$$c \equiv 3\phi * h + m \equiv -3X^4 + 6X^3 + 7X^2 - 4X - 5 \pmod{16}.$$

Bob decrypts by first computing

$$a \equiv f * c \equiv 4X^4 - 2X^3 - 5X^2 + 6X - 2 \pmod{16},$$

then

$$F_p * a \equiv X^2 - X + 1 \pmod{3}.$$

Therefore, Bob has obtained the message.

23.4.1 An Attack on NTRU

Let $h = h_{N-1}X^{N-1} + \cdots + h_0$. Form the

N×N matrix

H=(h0h1⋯hN−1hN−1h0⋯hN−2⋮⋮⋱⋮h1h2⋯h0).

If we represent f=fN−1XN−1+⋯+f0 and g=gN−1XN−1+⋯+g0 by the row vectors

f¯=(f0, …, fN−1) and g¯=(g0, …, gN−1),

then we see that $f\bar{}H \equiv g\bar{} \pmod{q}$.

Let I be the N×N identity matrix. Form the 2N×2N matrix

M=IH0qI.

Let L be the lattice generated by the rows of M. Since $g \equiv f*h \pmod{q}$, we can write g=f*h+qy for some polynomial y. Represent y as an N-dimensional row vector $y\bar{}$, so $(f\bar{}, y\bar{})$ is a 2N-dimensional row vector. Then

$(f^-, y^-)M=(f^-, g^-)$,

so (f^-, g^-) is in the lattice L (see Exercise 3). Since f and g have small coefficients, (f^-, g^-) is a small vector in the lattice L. Therefore, the secret information for the key can be represented as a short vector in a lattice. An attacker can try to apply a lattice reduction algorithm to find short vectors, and possibly obtain (f^-, g^-). Once the attacker has found f and g, the system is broken.

To stop lattice attacks, we need to make the lattice have high enough dimension that lattice reduction algorithms are inefficient. This is easily achieved by making N sufficiently large. However, if N is too large, the encryption and decryption algorithms become slow.

The suggested values of N were chosen to achieve security while keeping the cryptographic algorithms efficient.

Lattice reduction methods have the best success when the shortest vector is small (more precisely, small when compared to the 2Nth root of the determinant of the 2N-dimensional lattice). Improvements in the above lattice attack can be obtained by replacing I in the upper left block of M by αI for a suitably chosen real number α. This makes the resulting short vector $(\alpha f^{-}, g^{-})$ comparatively shorter and thus easier to find. The parameters in NTRU, especially the sizes of f and g, have been chosen so as to limit the effect of these lattice attacks.

So far, the NTRU cryptosystem appears to be strong; however, as with many new cryptosystems, the security is still being studied. If no successful attacks are found, NTRU will have the advantage of providing security comparable to RSA and other public key methods, but with smaller key size and with faster encryption and decryption times.

23.5 Another Lattice-Based Cryptosystem

Another lattice-based public key cryptosystem was developed by Goldreich, Goldwasser, and Halevi in 1997. Let L be a 300-dimensional lattice that is a subset of the points with integral coordinates in 300-dimensional real space R300.

The private key is a “good” basis of L, given by the columns of a 300×300 matrix G, where “good” means that the entries of G are small.

The public key is a “bad basis” of L, given by the columns of a 300×300

matrix B=GU, where U is a secret 300×300 matrix with integral entries and with determinant 1. The determinant condition implies that the entries of U−1 are also integers. “Bad” means that B has many large entries.

A message is a 300-dimensional vector $\vec{m}$ with integral entries, which is encrypted to obtain the ciphertext

$$\vec{c}=B\vec{m}+\vec{e},$$

where $\vec{e}$ is a 300-dimensional vector whose entries are chosen randomly from $\{0, \pm1, \pm2, \pm3\}$.

The decryption is carried out by computing

$$B-1G\lfloor G-1\vec{c}\rceil,$$

where $\lfloor \vec{v} \rceil$ for a vector means we round off each entry to the nearest integer (and .5 goes whichever way you specify). Why does this decryption work? First, $U = G - 1B$, so

$$G-1\vec{c} = G-1B\vec{m} + G-1\vec{e} = U\vec{m} + G-1\vec{e}.$$

Since U and $\vec{m}$ have integral entries, so does the $U\vec{m}$. Since G is good, the entries of $G-1\vec{e}$ tend to be small fractions, so they disappear in the rounding. Therefore, $\lfloor G-1\vec{c} \rceil$ probably equals $U\vec{m}$, so

$$B-1G\lfloor G-1\vec{c} \rceil = B-1GU\vec{m} = B-1GG-1B\vec{m} = \vec{m}.$$

Example

To keep things simple, we'll work in two-dimensional, rather than 300-dimensional, space. Let

$$G=5214 \text{and} B=GU=521417181617=1171248186$$

and

$$\vec{m}=5937.$$

Then

$$\vec{c}=11712481865937+1-1=114927960.$$

To decrypt, we first compute

$$\lfloor G-1\vec{c} \rceil = 2/9-1/9-1/185/18114927960=1669.331572.67=16691573.$$

Therefore, the decryption is

$$B-1G\lfloor G-1\vec{c} \rceil = U-1\lfloor G-1\vec{c} \rceil = 17-18-161716691573=5937=\vec{m}.$$

Suppose, instead, that we tried to decrypt by computing $\lfloor B-1\vec{c} \rceil$? In the present example,

$$B-1=43/9-62/9-9/213/2$$

and

$B-1\vec{c}=43/9-62/9-9/213/2114927960=212/326.$

This rounds off to (71, 26), which is nowhere close to the original message. The problem is that the entries of $B-1$ are much larger than those of $G-1$, so the small error introduced by $\vec{e}$ is amplified by $B-1$.

Attacking this system involves the **Closest Vector Problem:** Given a point P in Rn, find the point in L closest to P.

We have $B\vec{m} \in L$, and $\vec{c}$ is close to $B\vec{m}$ since it is moved off the lattice by the small vector $\vec{e}$.

For general lattices, the Closest Vector Problem is very hard. But it seems to be easier if the point is very close to a lattice point, which is the case in this

cryptosystem. So the actual level of security is not yet clear.

23.6 Post-Quantum Cryptography?

If a quantum computer is built (see Chapter 25), cryptosystems based on factorization or discrete logs will become less secure. An active area of current research involves designing systems that cannot be broken by a quantum computer. Some of the most promising candidates seem to be lattice-based systems since their security does not depend on the difficulty of computing discrete logs or factoring, and no attack with a quantum computer has been found. Similarly, the McEliece cryptosystem, which is based on error-correcting codes (see Section 24.10) and

is similar to the system in Section 23.5, seems to be a possibility.

One of the potential difficulties with using many of these lattice-based systems is the key size: In the system in Section 23.5, the public key B requires 3002=90000 integer entries. Many of the entries of B should be large, so let's say that we use 100 bits to specify each one. This means that the key requires 9×106 bits, much more than is used in current public key cryptosystems.

For more on this subject, see [Bernstein et al.].

23.7 Exercises

1. Find a reduced basis and a shortest nonzero vector in the lattice generated by the vectors (58, 19), (168, 55).

2.
    1. Find a reduced basis for the lattice generated by the vectors (53, 88), (107, 205).

    2. Find the vector in the lattice of part (a) that is closest to the vector (151, 33). (Remark: This is an example of the **closest vector problem**. It is fairly easy to solve when a reduced basis is known, but difficult in general. For cryptosystems based on the closest vector problem, see [Nguyen-Stern].)

3. Let v1, …, vn be linearly independent row vectors in Rn. Form the matrix M whose rows are the vectors vi. Let $\bar{a}$=(a1, …, an) be a row by v1, …, vn, and show that every vector in the lattice can be written in this way.

4. Let {v1, v2} be a basis of a lattice. Let a, b, c, d be integers with ad−bc=±1, and let

   w1=av1+bv2, w2=cv1+dv2.

   1. Show that

      v1=±(dw1−bw2), v2=±(−cw1+aw2).

   2. Show that {w1, w2} is also a basis of the lattice.

5. Let N be a positive integer.

   1. Show that if j+k ≡ i (mod N), then Xj+k −Xi is a multiple of XN−1.

   2. Let 0 ≤ i<N. Let a0, …, aN−1, b0, …, bN −1 be integers and let

      ci= ∑ j+k ≡ iajbk,

      where the sum is over pairs j, k with j +k ≡ i (mod N). Show that

      ciXi− ∑ j+k ≡ iajbkXj+k

      is a multiple of XN−1.

   3. Let f and g be polynomials of degree less than N. Let fg be the usual product of f and g and let f∗g be defined as in Section

23.4. Show that $fg-f*g$ is a multiple of $XN-1$.

6. Let N and p be positive integers. Suppose that there is a polynomial F(X) such that $f(X)*F(X) \equiv 1 \pmod{p}$. Show that $f(1) \equiv 0 \bmod p$. (Hint: Use Exercise 5(c).)

7.

    1. In the NTRU cryptosystem, suppose we ignore q and let $c=p\phi*h+m$. Show how an attacker can obtain the message quickly.

    2. In the NTRU cryptosystem, suppose q is a multiple of p. Show how an attacker can obtain the message quickly.

Chapter 24 Error Correcting Codes

In a good cryptographic system, changing one bit in the ciphertext changes enough bits in the corresponding plaintext to make it unreadable. Therefore, we need a way of detecting and correcting errors that could occur when ciphertext is transmitted.

Many noncryptographic situations also require error correction; for example, fax machines, computer hard drives, CD players, and anything that works with digitally represented data. Error correcting codes solve this problem.

Though coding theory (communication over noisy channels) is technically not part of cryptology (communication over nonsecure channels), in Section 24.10 we describe how error correcting codes can be used to construct a public key cryptosystem.

24.1 Introduction

All communication channels contain some degree of noise, namely interference caused by various sources such as neighboring channels, electric impulses, deterioration of the equipment, etc. This noise can interfere with data transmission. Just as holding a conversation in a noisy room becomes more difficult as the noise becomes louder, so too does data transmission become more difficult as the communication channel becomes noisier. In order to hold a conversation in a loud room, you either raise your voice, or you are forced to repeat yourself. The second method is the one

that will concern us; namely, we need to add some redundancy to the transmission in order for the recipient to be able to reconstruct the message. In the following, we give several examples of techniques that can be used. In each case, the symbols in the original message are replaced by *codewords* that have some redundancy built into them.

Example 1. (repetition codes)

Consider an alphabet {A, B, C, D}. We want to send a letter across a noisy channel that has a probability p=0.1 of error. If we want to send C, for example, then there is a 90% chance that the symbol received is C. This leaves too large a chance of error. Instead, we

repeat the symbol three times, thus sending CCC. Suppose an error occurs and the received word is CBC. We take the symbol that occurs most frequently as the message, namely C. The probability of the correct message being found is the probability that all three letters are correct plus the probability that exactly one of the three letters is wrong:

$$(0.9)3+3(0.9)2(0.1)=0.972,$$

which leaves a significantly smaller chance of error.

Two of the most important concepts for codes are error detection and error correction. If there are at most two errors, this repetition code allows us to detect that errors have occurred. If the

received message is CBC, then there could be either one error from CCC or two errors from BBB; we cannot tell which. If at most one error has occurred, then we can correct the error and deduce that the message was CCC. Note that if we used only two repetitions instead of three, we could detect the existence of one error, but we could not correct it (did CB come from BB or CC?).

This example was chosen to point out that error correcting codes can use arbitrary sets of symbols. Typically, however, the symbols that are used are mathematical objects such as integers mod a prime or binary strings. For example, we can replace the letters A, B, C, D by 2-bit strings: 00, 01, 10,

11. The preceding procedure (repeating three times) then gives us the codewords

000000, 010101, 101010, 111111.

Example 2. (parity check)

Suppose we want to send a message of seven bits. Add an eighth bit so that the number of nonzero bits is even. For example, the message 0110010 becomes 01100101, and the message 1100110 becomes 11001100. An error of one bit during transmission is immediately discovered since the message received will have an odd number of nonzero bits. However, it is impossible to tell which bit is incorrect, since an error in any bit could have yielded the odd number of nonzero bits. When an error

is detected, the best thing to do is resend the message.

Example 3. (two-dimensional parity code)

The parity check code of the previous example can be used to design a code that can correct an error of one bit. The two-dimensional parity code arranges the data into a two-dimensional array, and then parity bits are computed along each row and column.

To demonstrate the code, suppose we want to encode the 20 data bits 10011011001100101011. We arrange the bits into a 4×5 matrix

10011011001100101011

and calculate the parity bits along the rows and columns. We define the last bit in the lower right corner of the extended matrix by calculating the parity of the parity bits that were calculated along the columns. This results in the 5×6 matrix

100111011000110011010111011011.

Suppose that this extended matrix of bits is transmitted and that a bit error occurs at the bit in the third row and fourth column. The receiver arranges the received bits into a 5×6 matrix and obtains

100111011000110111010111011011.

The parities of the third row and fourth column are odd, so this locates the error as occurring at the third row and fourth column.

If two errors occur, this code can detect their existence. For example, if bit errors occur at the second and third bits of the second row, then the parity checks of the second and third columns will indicate the existence of two bit errors. However, in this case it is not possible to correct the errors, since there are several possible locations for them. For example, if the second and third bits of the fifth row were incorrect instead, then the parity checks would be the same as when these errors occurred in the second row.

Example 4. (Hamming [7, 4] code)

The original message consists of blocks of four binary bits. These are replaced

by codewords, which are blocks of seven bits, by multiplying (mod 2) on the right by the matrix

$$G=1000110010010100100110001111.$$

For example, the message 1100 becomes

$$(1,\ 1,\ 0,\ 0)1000110010010100100110001111 \equiv (1,\ 1,\ 0,\ 0,\ 0,\ 1,\ 1) \pmod 2.$$

Since the first four columns of G are the identity matrix, the first four entries of the output are the original message. The remaining three bits provide the redundancy that allows error detection and correction. In fact, as we'll see, we can easily correct an error if it affects only one of the seven bits in a codeword.

Suppose, for example, that the codeword 1100011 is sent but is received as 1100001. How do we detect and correct the error? Write G in the form [I4, P], where P is a 4×3 matrix. Form the matrix H=[PT, I3], where PT is the transpose of P. Multiply the message received times the transpose of H:

$$(1,\ 1,\ 0,\ 0,\ 0,\ 0,\ 1)$$
$$(11011001011010011 1001)T \equiv (1,\ 1,\ 0,\ 0,\ 0,\ 0,\ 1)$$
$$(11010101111110001 0001) \equiv (0,\ 1,\ 0) \pmod 2.$$

This is the 6th row of HT, which means there was an error in the 6th bit of the message received. Therefore, the correct codeword was 1100011. The first four bits give the original message 1100. If there had been no errors, the result of multiplying by HT would have been

(0, 0, 0), so we would have recognized that no correction was needed. This rather mysterious procedure will be explained when we discuss Hamming codes in Section 24.5. For the moment, note that it allows us to correct errors of one bit fairly efficiently.

The Hamming [7, 4] code is a significant improvement over the repetition code. In the Hamming code, if we want to send four bits of information, we transmit seven bits. Up to two errors can be detected and up to one error can be corrected. For a repetition code to achieve this level of error detection and correction, we need to transmit 12 bits in order to send a 4-bit message. Later, we'll express this

mathematically by saying that the code rate of this Hamming code is 4/7, while the code rate of the repetition code is 4/12=1/3. Generally, a higher code rate is better, as long as not too much error correcting capability is lost. For example, sending a 4-bit message as itself has a code rate of 1 but is unsatisfactory in most situations since there is no error correction capability.

Example 5. (ISBN code)

The International Standard Book Number (ISBN) provides another example of an error detecting code. The ISBN is a 10-digit codeword that is assigned to each book when it is published. For example, the first edition of this book had ISBN number

0-13-061814-4. The first digit represents the language that is used; 0 indicates English. The next two digits represent the publisher. For example, 13 is associated with Pearson/Prentice Hall. The next six numbers correspond to a book identity number that is assigned by the publisher. The tenth digit is chosen so that the ISBN number $a_1a_2 \cdots a_{10}$ satisfies

$$\sum_{j=1}^{10} ja_j \equiv 0 \pmod{11}.$$

Notice that the equation is done modulo 11. The first nine numbers $a_1a_2 \cdots a_9$ are taken from $\{0, 1, \cdots 9\}$ but a_{10} may be 10, in which case it is represented by the symbol X. Books published in 2007 or later use a 13-digit ISBN, which uses a slightly different sum and works mod

10.

Suppose that the ISBN number a1a2⋯a10 is sent over a noisy channel, or is written on a book order form, and is received as x1x2⋯x10. The ISBN code can detect a single error, or a double error that occurs due to the transposition of the digits. To accomplish this, the receiver calculates the weighted checksum

$$S = \sum_{j=1}^{10} jx_j \pmod{11}.$$

If $S \equiv 0 \pmod{11}$, then we do not detect any errors, though there is a small chance that an error occurred and was undetected. Otherwise, we have detected an error. However, we cannot correct it (see Exercise 2).

If $x_1x_2\cdots x_{10}$ is the same as $a_1a_2\cdots a_{10}$ except in one place x_k, we may write $x_k=a_k+e$ where $e \neq 0$. Calculating S gives

$$S=\sum_{j=1}^{10} ja_j+ke \equiv ke \pmod{11}.$$

Thus, if a single error occurs, we can detect it. The other type of error that can be *reliably* detected is when a_k and a_l have been transposed. This is one of the most common errors that occur when someone is copying numbers. In this case $x_l=a_k$ and $x_k=a_l$. Calculating S gives

$$S=\sum_{j=1}^{10} jx_j=\sum_{j=1}^{10} ja_j+(k-l)a_l+(l-k)a_k \pmod{11} \equiv (k-l)(a_l-a_k) \pmod{11}$$

If $a_l \neq a_k$, then the checksum is not equal to 0, and an error is detected.

Example 6. (Hadamard code)

This code was used by the *Mariner* spacecraft in 1969 as it sent pictures back to Earth. There are 64 codewords; 32 are represented by the rows of the 32×32 matrix

$$H=(1111\cdots11-11-1\cdots-1\vdots\ddots\vdots1-1-11\cdots-1).$$

The matrix is constructed as follows. Number the rows and columns from 0 to 31. To obtain the entry hij in the ith row and jth column, write i=a4a3a2a1a0 and j=b4b3b2b1b0 in binary. Then

$$hij=(-1)a0b0+a1b1+\cdots+a4b4.$$

For example, when i=31 and j=3, we have i=11111 and j=00011. Therefore, h31, 3=(−1)2=1.

The other 32 codewords are obtained by using the rows of $-H$. Note that the dot product of any two rows of H is 0, unless the two rows are equal, in which case the dot product is 32.

When *Mariner* sent a picture, each pixel had a darkness given by a 6-bit number. This was changed to one of the 64 codewords and transmitted. A received message (that is, a string of 1s and -1s of length 32) can be decoded (that is, corrected to a codeword) as follows. Take the dot product of the message with each row of H. If the message is correct, it will have dot product 0 with all rows except one, and it will have dot product ± 32 with that row. If the dot product is 32, the codeword is that row

of H. If it is −32, the codeword is the corresponding row of −H. If the message has one error, the dot products will all be ±2, except for one, which will be ±30. This again gives the correct row of H or −H. If there are two errors, the dot products will all be 0, ±2, ±4, except for one, which will be ±32, ±30, or ±28. Continuing, we see that if there are seven errors, all the dot products will be between −14 and 14, except for one between −30 and −16 or between 16 and 30, which yields the correct codeword. With eight or more errors, the dot products start overlapping, so correction is not possible. However, detection is possible for up to 15 errors, since it takes 16 errors to change one codeword to another.

This code has a relatively low code rate of 6/32, since it uses 32 bits to send a 6-bit message. However, this is balanced by a high error correction rate. Since the messages from *Mariner* were fairly weak, the potential for errors was high, so high error correction capability was needed. The other option would have been to increase the strength of the signal and use a code with a higher code rate and less error correction. The transmission would have taken less time and therefore potentially have used less energy. However, in this case, it turned out that using a weaker signal more than offset the loss in speed. This issue (technically known as **coding gain**) is an important engineering consideration in the choice of which code to use in a

given application.

24.2 Error Correcting Codes

A sender starts with a message and **encodes** it to obtain codewords consisting of sequences of symbols. These are transmitted over a noisy channel, depicted in Figure 24.1, to the receiver. Often the sequences of symbols that are received contain errors and therefore might not be codewords. The receiver must **decode**, which means correct the errors in order to change what is received back to codewords and then recover the original message.

Figure 24.1 Encoding and Decoding

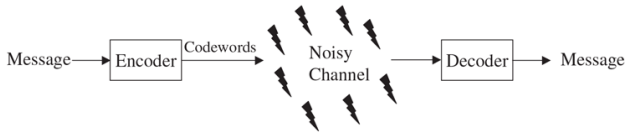


Figure 24.1 Full Alternative Text

The symbols used to construct the codewords belong to an alphabet. When the alphabet consists of the binary bits 0 and 1, the code is called a **binary code**. A code that uses sequences of three symbols, often represented as integers mod 3, is called a **ternary code**. In general, a code that uses an alphabet consisting of q symbols is called a **q-ary code**.

Definition

Let A be an alphabet and let An denote the set of n-tuples of elements of A. A **code of length** n is a nonempty subset

of An.

The n-tuples that make up a code are called **codewords**, or code vectors. For example, in a binary repetition code where each symbol is repeated three times, the alphabet is the set A={0, 1} and the code is the set
{(0, 0, 0), (1, 1, 1)} ⊂ A3.

Strictly speaking, the codes in the definition are called **block codes**. Other codes exist where the codewords can have varying lengths. These will be mentioned briefly at the end of this chapter, but otherwise we focus only on block codes.

For a code that is a random subset of An, decoding could be a time-

consuming procedure. Therefore, most useful codes are subsets of An satisfying additional conditions. The most common is to require that A be a finite field, so that An is a vector space, and require that the code be a subspace of this vector space. Such codes are called *linear* and will be discussed in Section 24.4.

For the rest of this section, however, we work with arbitrary, possibly nonlinear, codes. We always assume that our codewords are n-dimensional vectors.

In order to decode, it will be useful to put a measure on how close two vectors are to each other. This is provided by the Hamming distance. Let u, v be two vectors in An. The **Hamming distance**

d(u, v) is the number of places where the two vectors differ. For example, if we use binary vectors and have the vectors u=(1, 0, 1, 0, 1, 0, 1, 0) and v=(1, 0, 1, 1, 1, 0, 0, 0), then u and v differ in two places (the 4th and the 7th) so d(u, v)=2. As another example, suppose we are working with the usual English alphabet. Then d(fourth, eighth)=4 since the two strings differ in four places.

The importance of the Hamming distance d(u, v) is that it measures the minimum number of “errors” needed for u to be changed to v. The following gives some of its basic properties.

Proposition

d(u, v) is a metric on An, which means that it satisfies

1. d(u, v) ≥ 0, and d(u, v)=0 if and only if u=v
2. d(u, v)=d(v, u) for all u, v
3. d(u, v) ≤ d(u, w)+d(w, v) for all u, v, w.

The third property is often called the triangle inequality.

Proof. (1) d(u, v)=0 is exactly the same as saying that u and v differ in no places, which means that u=v. Part (2) is obvious. For part (3), observe that if u and v differ in a place, then either u and w differ at that place, or v and w differ at that place, or both. Therefore, the number of places where u and v differ is less than or equal to the number of places where u and w differ, plus the

number of places where v and w differ.

For a code C, one can calculate the Hamming distance between any two distinct codewords. Out of this table of distances, there is a minimum value d(C), which is called the **minimum distance** of C. In other words,

$$d(C)=\min\{ d(u, v) | u, v \in C, u \neq v \}.$$

The minimum distance of C is very important number, since it gives the smallest number of errors needed to change one codeword into another.

When a codeword is transmitted over a noisy channel, errors are introduced into some of the entries of the vector. We correct these errors by finding the codeword whose Hamming distance

from the received vector is as small as possible. In other words, we change the received vector to a codeword by changing the fewest places possible. This is called **nearest neighbor decoding**.

We say that a code can **detect** up to s errors if changing a codeword in at most s places cannot change it to another codeword. The code can **correct** up to t errors if, whenever changes are made at t or fewer places in a codeword c, then the closest codeword is still c. This definition says nothing about an efficient algorithm for correcting the errors. It simply requires that nearest neighbor decoding gives the correct answer when there are at most t errors. An important

result from the theory of error correcting codes is the following.

Theorem

1. A code C can detect up to s errors if $d(C) \geq s+1$. 2. A code C can correct up to t errors if $d(C) \geq 2t+1$.

Proof.

1. Suppose that $d(C) \geq s+1$. If a codeword c is sent and s or fewer errors occur, then the received message r cannot be a different codeword. Hence, an error is detected.

2. Suppose that $d(C) \geq 2t+1$. Assume that the codeword c is sent and the received word r has t or fewer errors; that is, $d(c, r) \leq t$. If c1 is any other codeword besides c, we claim that $d(c1, r) \geq t+1$. To see this, suppose that $d(c1, r) \leq t$. Then, by applying the triangle inequality, we have

   $$2t+1 \leq d(C) \leq d(c, c1) \leq d(c, r)+d(c1, r) \leq t+t=2t.$$

   This is a contradiction, so $d(c1, r) \geq t+1$. Since r

has t or fewer errors, nearest neighbor decoding successfully decodes r to c.

How does one find the nearest neighbor? One way is to calculate the distance between the received message r and each of the codewords, then select the codeword with the smallest Hamming distance. In practice, this is impractical for large codes. In general, the problem of decoding is challenging, and considerable research effort is devoted to looking for fast decoding algorithms. In later sections, we'll discuss a few decoding techniques that have been developed for special classes of codes.

Before continuing, it is convenient to introduce some notation.

Notation

A code of length n, with M codewords, and with minimum distance d=d(C), is called an (n,M,d) **code**.

When we discuss linear codes, we'll have a similar notation, namely, an [n, k, d] code. Note that this latter notation uses square brackets, while the present one uses curved parentheses. (These two similar notations cause less confusion than one might expect!) The relation is that an [n, k, d] binary linear code is an (n, 2k, d) code.

The binary repetition code {(0, 0, 0), (1, 1, 1)} is a (3, 2, 3) code. The Hadamard code of Exercise 6, Section 24.1, is a (32, 64, 16) code (it

could correct up to 7 errors because $16 \geq 2 \cdot 7+1$).

If we have a q-ary (n, M, d) code, then we define the **code rate**, or **information rate**, R by

$$R=\log_q \frac{M}{n}.$$

For example, for the Hadamard code, $R=\log_2 (64)/32=6/32$. The code rate represents the ratio of the number of input data symbols to the number of transmitted code symbols. It is an important parameter to consider when implementing real-world systems, as it represents what fraction of the bandwidth is being used to transmit actual data. The code rate was mentioned in Examples 4 and 6 in Section 24.1. A few limitations on the

code rate will be discussed in Section 24.3.

Given a code, it is possible to construct other codes that are essentially the same. Suppose that we have a codeword c that is expressed as $c=(c1, c1, \cdots, cn)$. Then we may define a positional permutation of c by permuting the order of the entries of c. For example, the new vector $c'=(c2, c3, c1)$ is a positional permutation of $c=(c1, c2, c3)$. Another type of operation that can be done is a symbol permutation. Suppose that we have a permutation of the q-ary symbols. Then we may fix a position and apply this permutation of symbols to that fixed position for every codeword. For example, suppose that

we have the following permutation of the ternary symbols {0→2, 1→0, 2→1}, and that we have the following codewords: (0, 1, 2), (0, 2, 1), and (2, 0, 1). Then applying the permutation to the second position of all of the codewords gives the following vectors: (0, 0, 2), (0, 1, 1), and (2, 2, 1).

Formally, we say that two codes are **equivalent** if one code can be obtained from the other by a series of the following operations:

1. Permuting the positions of the code
2. Permuting the symbols appearing in a fixed position of all codewords

It is easy to see that all codes equivalent to an (n, M, d) code are also (n, M, d)

codes. However, for certain choices of n, M, d, there can be several inequivalent (n, M, d) codes.

24.3 Bounds on General Codes

We have shown that an (n, M, d) code can correct t errors if $d \geq 2t+1$. Hence, we would like the minimum distance d to be large so that we can correct as many errors as possible. But we also would like for M to be large so that the code rate R will be as close to 1 as possible. This would allow us to use bandwidth efficiently when transmitting messages over noisy channels. Unfortunately, increasing d tends to increase n or decrease M.

In this section, we study the restrictions on n, M, and d without worrying about

practical aspects such as whether the codes with good parameters have efficient decoding algorithms. It is still useful to have results such as the ones we'll discuss since they give us some idea of how good an actual code is, compared to the theoretical limits.

First, we treat upper bounds for M in terms of n and d. Then we show that there exist codes with M larger than certain lower bounds. Finally, we see how some of our examples compare with these bounds.

24.3.1 Upper Bounds

Our first result was given by R. Singleton in 1964 and is known as the **Singleton bound**.

Theorem

Let C be a q-ary (n, M, d) code. Then

$M \leq q^{n-d+1}$.

Proof. For a codeword $c=(a_1, \ldots, a_n)$, let $c'=(a_d, \ldots, a_n)$. If $c_1 \neq c_2$ are two codewords, then they differ in at least d places. Since c_1' and c_2' are obtained by removing $d-1$ entries from c_1 and c_2, they must differ in at least one place, so $c_1' \neq c_2'$. Therefore, the number M of codewords c equals the number of vectors c' obtained in this way. There are at most q^{n-d+1} vectors c' since there are $n-d+1$ positions in these vectors. This implies that $M \leq q^{n-d+1}$, as desired.

Corollary

The code rate of a q-ary (n, M, d) code is at most 1−d−1n.

Proof. The corollary follows immediately from the definition of code rate.

The corollary implies that if the **relative minimum distance** d/n is large, the code rate is forced to be small.

A code that satisfies the Singleton bound with equality is called an **MDS code** (maximum distance separable). The Singleton bound can be rewritten as qd ≤ qn+1/M, so an MDS code has the largest possible value of d for a given n and M. The Reed-Solomon codes (Section 24.9) are an important class of MDS codes.

Before deriving another upper bound, we need to introduce a geometric interpretation that is useful in error correction. A **Hamming sphere** of radius t centered at a codeword c is denoted by B(c, t) and is defined to be all vectors that are at most a Hamming distance of t from the codeword c. That is, a vector u belongs to the Hamming sphere B(c, t) if d(c, u) ≤ t. We calculate the number of vectors in B(c, t) in the following lemma.

Lemma

A sphere B(c, r) in n-dimensional q-ary space has

$$n0+n1(q−1)+n2(q−1)2+⋯+nr(q−1)r$$

elements.

Proof. First we calculate the number of vectors that are a distance 1 from c. These vectors are the ones that differ from c in exactly one location. There are n possible locations and q−1 ways to make an entry different. Thus the number of vectors that have a Hamming distance of 1 from c is n(q−1). Now let's calculate the number of vectors that have Hamming distance m from c. There are nm ways in which we can choose m locations to differ from the values of c. For each of these m locations, there are q−1 choices for symbols different from the corresponding symbol from c. Hence, there are

nm(q−1)m

vectors that have a Hamming distance of m from c. Including the vector c itself, and using the identity n0=1, we get the result:

$$n0+n1(q-1)+n2(q-1)2+\cdots+nr(q-1)r.$$

We may now state the **Hamming bound**, which is also called the **sphere packing bound**.

Theorem

Let C be a q-ary (n, M, d) code with $d \geq 2t+1$. Then

$$M \leq qn \sum j=0t(nj)(q-1)j.$$

Proof. Around each codeword c we place a Hamming sphere of radius t. Since the minimum distance of the code is $d \geq 2t+1$, these spheres do not

overlap. The total number of vectors in all of the Hamming spheres cannot be greater than qn. Thus, we get

(number of codewords)×(number of elements per sphere)=M $\sum_{j=0}^{t} \binom{n}{j} (q-1)^j \leq q^n$.

This yields the desired inequality for M.

An (n, M, d) code with d=2t+1 that satisfies the Hamming bound with equality is called a **perfect code**. A perfect t-error correcting code is one such that the M Hamming spheres of radius t with centers at the codewords cover the entire space of q-ary n-tuples. The Hamming codes (Section 24.5) and the Golay code G23 (Section 24.6) are perfect. Other examples of perfect codes are the trivial (n, qn, 1) code obtained by taking all n-tuples, and the binary

repetition codes of odd length (Exercise 15).

Perfect codes have been studied a lot, and they are interesting from many viewpoints. The complete list of perfect codes is now known. It includes the preceding examples, plus a ternary [11, 6, 5] code constructed by Golay. We leave the reader a caveat. A name like *perfect codes* might lead one to assume that perfect codes are the best error correcting codes. This, however, is not true, as there are error correcting codes, such as Reed-Solomon codes, that are not perfect codes yet have better error correcting capabilities for certain situations than perfect codes.

24.3.2 Lower Bounds

One of the problems central to the theory of error correcting codes is to find the largest code of a given length and given minimum distance d. This leads to the following definition.

Definition

Let the alphabet A have q elements. Given n and d with $d \leq n$, the largest M such that an (n, M, d) code exists is denoted Aq(n, d).

We can always find at least one (n, M, d) code: Fix an element a0 of A. Let C be the set of all vectors (a, a, …, a, a0, …, a0) (with d copies of a and n−d copies of a0) with $a \in A$.

There are q such vectors, and they are at distance d from each other, so we have an (n, q, d) code. This gives the trivial lower bound $A_q(n, d) \geq q$. We'll obtain much better bounds later.

It is easy to see that $A_q(n, 1)=q^n$: When a code has minimum distance d=1, we can take the code to be all q-ary n-tuples. At the other extreme, $A_q(n, n)=q$ (Exercise 7).

The following lower bound, known as the **Gilbert-Varshamov bound**, was discovered in the 1950s.

Theorem

Given n, d with $n \geq d$, there exists a q-ary (n, M, d) code with

$$M \geq q^n \sum_{j=0}^{d-1} \binom{n}{j} (q-1)^j.$$

This means that

$$A_q(n, d) \geq q^n \sum_{j=0}^{d-1} \binom{n}{j} (q-1)^j.$$

Proof. Start with a vector c_1 and remove all vectors in A^n (where A is an alphabet with q symbols) that are in a Hamming sphere of radius $d-1$ about that vector. Now choose another vector c_2 from those that remain. Since all vectors with distance at most $d-1$ from c_1 have been removed, $d(c_2, c_1) \geq d$. Now remove all vectors that have distance at most $d-1$ from c_2, and choose c_3 from those that remain. We cannot have $d(c_3, c_1) \leq d-1$ or $d(c_3, c_2) \leq d-1$, since all vectors satisfying these inequalities have been removed. Therefore, $d(c_3, c_i) \geq d$ for

i=1, 2. Continuing in this way, choose c4, c5, …, until there are no more vectors.

The selection of a vector removes at most

$$\sum_{j=0}^{d-1} n^j (q-1)^j$$

vectors from the space. If we have chosen M vectors c1, …, cM, then we have removed at most

$$M \sum_{j=1}^{d-1} n^j (q-1)^j$$

vectors, by the preceding lemma. We can continue until all qn vectors are removed, which means we can continue at least until

$$M \sum_{j=1}^{d-1} n^j (q-1)^j \geq q^n.$$

Therefore, there exists a code

{c1, …, cM} with M satisfying the preceding inequality.

Since Aq(n, d) is the largest such M, it also satisfies the inequality.

There is one minor technicality that should be mentioned. We actually have constructed an (n, M, e) code with e ≥ d. However, by modifying a few entries of c2 if necessary, we can arrange that d(c2, c1)=d. The remaining vectors are then chosen by the above procedure. This produces a code where the minimal distance is exactly d.

If we want to send codewords with n bits over a noisy channel, and there is a probability p that any given bit will be corrupted, then we expect the number of

errors to be approximately pn when n is large. Therefore, we need an (n, M, d) code with $d>2pn$. We therefore need to consider (n, M, d) codes with $d/n \approx x>0$, for some given $x>0$. How does this affect M and the code rate?

Here is what happens. Fix q and choose x with $0<x<1-1/q$. The **asymptotic Gilbert-Varshamov bound** says that there is a sequence of q-ary (n, M, d) codes with $n \to \infty$ and $d/n \to x$ such that the code rate approaches a limit $\geq Hq(x)$, where

$$Hq(x)=1-x\log_q(q-1)+x\log_q(x)+(1-x)\log_q(1-x).$$

The graph of H2(x) is as in Figure 24.2. Of course, we would like to have codes with high error correction (that is, high x), and with high code rate (=k/n). The

asymptotic result says that there are codes with error correction and code rate good enough to lie arbitrarily close to, or above, the graph.

Figure 24.2 The Graph of H2(x)

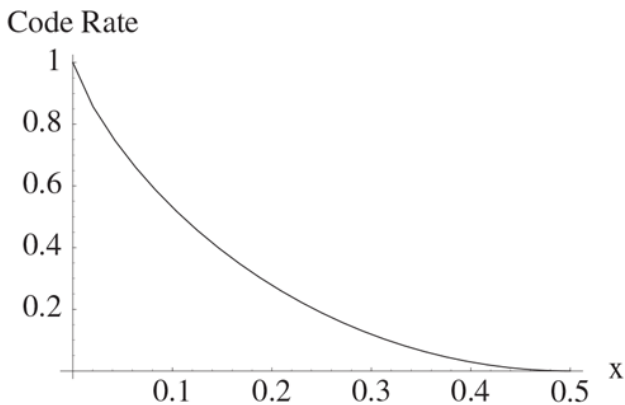


Figure 24.2 Full Alternative Text

The existence of certain sequences of codes having code rate limit strictly larger than Hq(x) (for certain x and q) was proved in 1982 by Tsfasman, Vladut, and Zink using Goppa codes

arising from algebraic geometry.

Examples

Consider the binary repetition code C of length 3 with the two vectors (0, 0, 0) and (1, 1, 1). It is a (3, 2, 3) code. The Singleton bound says that $2=M \leq 2$, so C is an MDS code. The Hamming bound says that

$$2=M \leq 2330+31=2,$$

so C is also perfect. The Gilbert-Varshamov bound says that there exists a (3, M, 3) binary code with

$$M \geq 2330+31+32=87,$$

which means $M \geq 2$.

The Hamming [7, 4] code has M=16

and d=3, so it is a (7, 16, 3) code. The Singleton bound says that 16=M ≤ 25, so it is not an MDS code. The Hamming bound says that

16=M ≤ 2770+71=16,

so the code is perfect. The Gilbert-Varshamov bound says that there exists a (7, M, 3) code with

M ≥ 2770+71+72=12829 ≈ 4.4,

so the Hamming code is much better than this lower bound. Codes that have efficient error correction algorithms and also exceed the Gilbert-Varshamov bound are currently relatively rare.

The Hadamard code from Section 24.1 is a binary (because there are two symbols) (32, 64, 16) code. The

Singleton bound says that 64=M ≤ 217, so it is not very sharp in this case. The Hamming bound says that

64=M ≤ 232 ∑ j=0732j ≈ 951.3.

The Gilbert-Varshamov bound says there exists a binary (32, M, 16) code with

M ≥ 232 ∑ j=01532j ≈ 2.3.

24.4 Linear Codes

When you are having a conversation with a friend over a cellular phone, your voice is turned into digital data that has an error correcting code applied to it before it is sent. When your friend receives the data, the errors in transmission must be accounted for by decoding the error correcting code. Only after decoding are the data turned into sound that represents your voice.

The amount of delay it takes for a packet of data to be decoded is critical in such an application. If it took several seconds, then the delay would become aggravating and make holding a

conversation difficult.

The problem of efficiently decoding a code is therefore of critical importance. In order to decode quickly, it is helpful to have some structure in the code rather than taking the code to be a random subset of An. This is one of the primary reasons for studying linear codes. For the remainder of this chapter, we restrict our attention to linear codes.

Henceforth, the alphabet A will be a finite field F. For an introduction to finite fields, see Section 3.11. For much of what we do, the reader can assume that F is Z2={0, 1} = the integers mod 2, in which case we are working with binary vectors. Another concrete example of a finite field is Zp = the

integers mod a prime p. For other examples, see Section 3.11. In particular, as is pointed out there, F must be one of the finite fields GF(q); but the present notation is more compact. Since we are working with arbitrary finite fields, we'll use "=" instead of " ≡ " in our equations. If you want to think of F as being Z2, just replace all equalities between elements of F with congruences mod 2.

The set of n-dimensional vectors with entries in F is denoted by Fn. They form a vector space over F. Recall that a subspace of Fn is a nonempty subset S that is closed under linear combinations, which means that if s1, s2 are in S and a1, a2 are in F, then $a1s1+a2s2 \in S$. By

taking a1=a2=0, for example, we see that (0, 0, , …, 0) ∈ S.

Definition

A **linear code** of dimension k and length n over a field F is a k-dimensional subspace of Fn. Such a code is called an [n,k] **code**. If the minimum distance of the code is d, then the code is called an [n,k,d] **code**.

When F=Z2, the definition can be given more simply. A binary code of length n and dimension k is a set of 2k binary n-tuples (the codewords) such that the sum of any two codewords is always a codeword.

Many of the codes we have met are

linear codes. For example, the binary repetition code {(0, 0, 0), (1, 1, 1)} is a one-dimensional subspace of Z23. The parity check code from Exercise 2 in Section 24.1 is a linear code of dimension 7 and length 8. It consists of those binary vectors of length 8 such that the sum of the entries is 0 mod 2. It is not hard to show that the set of such vectors forms a subspace. The vectors

(1, 0, 0, 0, 0, 0, 0, 1), (0, 1, 0, 0, 0, 0, 0, 1), …, (0, 0, 0, 0, 0, 0, 1, 1)

form a basis of this subspace. Since there are seven basis vectors, the subspace is seven-dimensional.

The Hamming [7, 4] code from Example 4 of Section 24.1 is a linear code of dimension 4 and length 7. Every codeword is a linear combination of the

four rows of the matrix G. Since these four rows span the code and are linearly independent, they form a basis.

The ISBN code (Example 5 of Section 24.1) is not linear. It consists of a set of 10-dimensional vectors with entries in Z11. However, it is not closed under linear combinations since X is not allowed as one of the first nine entries.

Let C be a linear code of dimension k over a field F. If F has q elements, then C has qk elements. This may be seen as follows. There is a basis of C with k elements; call them v1, …, vk. Every element of C can be written uniquely in the form a1v1+⋯+akvk, with a1, …, ak ∈ F. There are q choices for each ai and there are k numbers ai. This

means there are qk elements of C, as claimed. Therefore, an [n, k, d] linear code is an (n, qk, d) code in the notation of Section 24.2.

For an arbitrary, possibly nonlinear, code, computing the minimum distance could require computing d(u, v) for every pair of codewords. For a linear code, the computation is much easier. Define the **Hamming weight** wt(u) of a vector u to be the number of nonzero places in u. It equals d(u, 0), where 0 denotes the vector (0, 0, …, 0).

Proposition

Let C be a linear code. Then d(C) equals the smallest Hamming weight of all nonzero code vectors: d(C)=min{wt(u) |

$0 \neq u \in C\}$.

Proof. Since wt(u)=d(u, 0) is the distance between two codewords, we have wt(u) ≥ d(C) for all codewords u. It remains to show that there is a codeword with weight equal to d(C). Note that d(v, w)=wt(v−w) for any two vectors v, w. This is because an entry of v−w is nonzero, and hence gets counted in wt(v−w), if and only if v and w differ in that entry. Choose v and w to be distinct codewords such that d(v, w)=d(C). Then wt(v−w)=d(C), so the minimum weight of the nonzero codewords equals d(C).

To construct a linear [n, k] code, we have to construct a k-dimensional subspace of Fn. The easiest way to do

this is to choose k linearly independent vectors and take their span. This can be done by choosing a k×n **generating matrix** G of rank k, with entries in F. The set of vectors of the form vG, where v runs through all row vectors in Fk, then gives the subspace.

For our purposes, we'll usually take G=[Ik, P], where Ik is the k×k identity matrix and P is a k×(n−k) matrix. The rows of G are the basis for a k-dimensional subspace of the space of all vectors of length n. This subspace is our linear code C. In other words, every codeword is uniquely expressible as a linear combination of rows of G. If we use a matrix G=[Ik, P] to construct a code, the first k columns determine the

codewords. The remaining n−k columns provide the redundancy.

The code in the second half of Example 1, Section 24.1, has

$$G=101010010101.$$

The codewords 101010 and 010101 appear as rows in the matrix and the codeword 111111 is the sum of these two rows. This is a [6, 2] code.

The code in Example 2 has

$$G=10000001010000010010000100010001000010010000010100000011.$$

For example, the codeword 11001001 is the sum mod 2 of the first, second, and fifth rows, and hence is obtained by multiplying (1, 1, 0, 0, 1, 0, 0) times G. This is an [8, 7] code.

In Exercise 4, the matrix G is given in the description of the code. As you can guess from its name, it is a [7, 4] code.

As mentioned previously, we could start with any k×n matrix of rank k. Its rows would generate an [n, k] code. However, row and column operations can be used to transform the matrix to the form of G we are using, so we usually do not work with the more general situation. A code described by a matrix G=[Ik, P] as before is said to be **systematic**. In this case, the first k bits are the **information symbols** and the last n−k symbols are the **check symbols**.

Suppose we have G=[Ik, P] as the generating matrix for a code C. Let

$H=[-P^T, I_{n-k}]$,

where P^T is the transpose of P. In Exercise 4 of Section 24.1, this is the matrix that was used to correct errors. For Exercise 2, we have $H=[1, 1, 1, 1, 1, 1, 1, 1]$. Note that in this case a binary string v is a codeword if and only if the number of nonzero bits is even, which is the same as saying that its dot product with H is zero. This can be rewritten as $vH^T=0$, where H^T is the transpose of H.

More generally, suppose we have a linear code $C \subset F^n$. A matrix H is called a **parity check matrix** for C if H has the property that a vector $v \in F^n$ is in C if and only if $vH^T=0$. We have the following useful result.

Theorem

If G=[Ik, P] is the generating matrix for a code C, then H=[−PT, In−k] is a parity check matrix for C.

Proof. Consider the ith row of G, which has the form

$$v_i=(0, \ldots, 1, \ldots, 0, p_i, 1, \ldots, p_i, n-k),$$

where the 1 is in the ith position. This is a vector of the code C. The jth column of HT is the vector

$$(-p_{1, j}, \ldots, -p_{n-k, j}, 0, \ldots, 1, \ldots, 0),$$

where the 1 is in the (n−k+j)th position. To obtain the jth element of viHT, take the dot product of these two vectors, which yields

$$1\cdot(-p_{i, j})+p_{i, j}\cdot 1=0.$$

Therefore, HT annihilates every row vi of G. Since every element of C is a sum of rows of G, we find that vHT=0 for all $v \in C$.

Recall the following fact from linear algebra: The left null space of an $m \times n$ matrix of rank r has dimension $n-r$. Since HT contains I_{n-k} as a submatrix, it has rank $n-k$. Therefore, its left null space has dimension k. But we have just proved that C is contained in this null space. Since C also has dimension k, it must equal the null space, which is what the theorem claims.

We now have a way of detecting errors: If v is received during a transmission and $vHT \neq 0$, then there is an error. If vHT=0, we cannot conclude that there is

no error, but we do know that v is a codeword. Since it is more likely that no errors occurred than enough errors occurred to change one codeword into another codeword, the best guess is that an error did not occur.

We can also use a parity check matrix to make the task of decoding easier. First, let's look at an example.

Example

Let C be the binary linear code with generator matrix

G=10110110.

We are going to make a table of all binary vectors of length 4 according to the following procedure. First, list the

four elements of the code in the first row, starting with (0, 0, 0, 0). Then, among the 12 remaining vectors, choose one of smallest weight (there might be several choices). Add this vector to the first row to obtain the second row. From the remaining eight vectors, again choose one with smallest weight and add it to the first row to obtain the third row. Finally, choose a vector with smallest weight from the remaining four vectors, add it to the first row, and obtain the fourth row. We obtain the following:

(0, 0, 0, 0)(1, 0, 1, 1)(0, 1, 1, 0)(1, 1, 0, 1)(1, 0, 0, 0)(0, 0, 1, 1)
(1, 1, 1, 0)(0, 1, 0, 1)(0, 1, 0, 0)(1, 1, 1, 1)(0, 0, 1, 0)(1, 0, 0, 1)
(0, 0, 0, 1)(1, 0, 1, 0)(0, 1, 1, 1)(1, 1, 0, 0).

This can be used as a decoding table. When we receive a vector, find it in the

table. Decode by changing the vector to the one at the top of its column. The error that is removed is first element of its row. For example, suppose we receive (0, 1, 0, 1). It is the last element of the second row. Decode it to (1, 1, 0, 1), which means removing the error (1, 0, 0, 0). In this small example, this is not exactly the same as nearest neighbor decoding, since (0, 0, 1, 0) decodes as (0, 1, 1, 0) when it has an equally close neighbor (0, 0, 0, 0). The problem is that the minimum distance of the code is 2, so general error correction is not possible. However, if we use a code that can correct up to t errors, this procedure correctly decodes all vectors that are distance at most t from a codeword.

In a large example, finding the vector in the table can be tedious. In fact, writing the table can be rather difficult (that's why we used such a small example). This is where a parity check matrix H comes to the rescue.

The first vector v in a row is called the **coset leader**. Let r be any vector in the same row as v. Then r=v+c for some codeword c, since this is how the table was constructed. Therefore,

$$rHT=vHT+cHT=vHT,$$

since cHT=0 by the definition of a parity check matrix. The vector S(r)=rHT is called the **syndrome** of r. What we have shown is that two vectors in the same row have the same syndrome. Replace the preceding table with the following

much smaller table.

This table may be used for decoding as follows. For a received vector r, calculate its syndrome S(r)=rHT. Find this syndrome on the list and subtract the corresponding coset leader from r. This gives the same decoding as above.

For example, if r=(0, 1, 0, 1), then

$$S(r)=(0,\ 1,\ 0,\ 1)11101001=(1,\ 1).$$

This is the syndrome for the second row. Subtract the coset leader (1, 0, 0, 0) from r to obtain the codeword (1, 1, 0, 1).

We now consider the general situation. The method of the example leads us to two definitions.

Definition

Let C be a linear code and let u be an n-dimensional vector. The set u+C given by

$$u+C=\{u+c \mid c \in C\}$$

is called a **coset** of C.

It is easy to see that if $v \in u+C$, then the sets $v+C$ and $u+C$ are the same (Exercise 9).

Definition

A vector having minimum Hamming weight in a coset is called a **coset leader**.

The **syndrome** of a vector u is defined to be $S(u)=uH^T$. The following lemma allows us to determine the cosets easily.

Lemma

Two vectors u and v belong to the same coset if and only if they have the same syndrome.

Proof. Two vectors u and v to belong to the same coset if and only if their difference belongs to the code C; that is, $u-v \in C$. This happens if and only if $(u-v)HT=0$, which is equivalent to $S(u)=uHT=vHT=S(v)$.

Decoding can be achieved by building a syndrome lookup table, which consists of the coset leaders and their corresponding syndromes. With a syndrome lookup table, we can decode with the following steps:

1. For a received vector r, calculate its syndrome $S(r)=rHT$.

2. Next, find the coset leader with the same syndrome as S(r). Call the coset leader c0.

3. Decode r as $r-c0$.

Syndrome decoding requires significantly fewer steps than searching for the nearest codeword to a received vector. However, for large codes it is still too inefficient to be practical. In general, the problem of finding the nearest neighbor in a general linear code is hard; in fact, it is what is known as an NP-complete problem. However, for certain special types of codes, efficient decoding is possible. We treat some examples in the next few sections.

24.4.1 Dual Codes

The vector space Fn has a dot product, defined in the usual way:

$$(a1, \ldots, an)\cdot(b0, \ldots, bn)=a0b0+\cdots+anbn.$$

For example, if F=Z2, then

$(0, 1, 0, 1, 1, 1)\cdot(0, 1, 0, 1, 1, 1)=0,$

so we find the possibly surprising fact that the dot product of a nonzero vector with itself can sometimes be 0, in contrast to the situation with real numbers. Therefore, the dot product does not tell us the length of a vector. But it is still a useful concept.

If C is a linear [n, k] code, define the **dual code**

$C\perp=\{u\in Fn \mid u\cdot c=0 \text{ for all } c\in C\}.$

Proposition

If C is a linear [n, k] code with generating matrix $G=[Ik, P]$, then $C\perp$ is a linear [n, n−k] code with generating matrix $H=[-PT, In-k]$. Moreover, G is a

parity check matrix for $C^{\perp}$.

Proof. Since every element of C is a linear combination of the rows of G, a vector u is in $C^{\perp}$ if and only if $uG^T=0$. This means that $C^{\perp}$ is the left null space of G^T. Also, we see that G is a parity check matrix for $C^{\perp}$. Since G has rank k, so does G^T. The left null space of G^T therefore has dimension $n-k$, so $C^{\perp}$ has dimension $n-k$. Because H is a parity check matrix for C, and the rows of G are in C, we have $GH^T=0$. Taking the transpose of this relation, and recalling that transpose reverses order ($(AB)^T=B^TA^T$), we find $HG^T=0$. This means that the rows of H are in the left null space of G^T; therefore, in $C^{\perp}$. Since H has rank $n-k$, the span of its

rows has dimension n−k, which is the same as the dimension of $C^{\perp}$. It follows that the rows of H span $C^{\perp}$, so H is a generating matrix for $C^{\perp}$.

A code C is called **self-dual** is $C=C^{\perp}$. The Golay code G24 of Section 24.6 is an important example of a self-dual code.

Example

Let $C=\{(0, 0, 0), (1, 1, 1)\}$ be the binary repetition code. Since $u\cdot(0, 0, 0)=0$ for every u, a vector u is in $C^{\perp}$ if and only if $u\cdot(1, 1, 1)=0$. This means that $C^{\perp}$ is a parity check code: $(a1, a2, a3) \in C^{\perp}$ if and only if $a1+a2+a3=0$.

Example

Let C be the binary code with generating matrix

G=10010110.

The proposition says that C⊥ has generating matrix

H=01101001.

This is G with the rows switched, so the rows of G and the rows of H generate the same subspace. Therefore, C=C⊥, which says that C is self-dual.

24.5 Hamming Codes

The Hamming codes are an important class of single error correcting codes that can easily encode and decode. They were originally used in controlling errors in long-distance telephone calls. Binary Hamming codes have the following parameters:

1. Code length: $n = 2^m - 1$
2. Dimension: $k = 2^m - m - 1$
3. Minimum distance: $d = 3$

The easiest way to describe a Hamming code is through its parity check matrix. For a binary Hamming code of length $n = 2^m - 1$, first construct an $m \times n$ matrix

whose columns are all nonzero binary m-tuples. For example, for a [7, 4] binary Hamming code we take m=3, so n=7 and k=4, and start with

10101010111001100011111.

In order to obtain a parity check matrix for a code in systematic form, we move the appropriate columns to the end so that the matrix ends with the m×m identity matrix. The order of the other columns is irrelevant. The result is the parity check matrix H for a Hamming [n, k] code. In our example, we move the 4th, 2nd, and 1st columns to the end to obtain

H=11011001011010011111001,

which is the matrix H from Exercise 3.

We can easily calculate a generator matrix G from the parity check matrix H. Since Hamming codes are single error correcting codes, the syndrome method for decoding can be simplified. In particular, the error vector e is allowed to have weight at most 1, and therefore will be zero or will have all zeros except for a single 1 in the jth position.

The Hamming decoding algorithm, which corrects up to one bit error, is as follows:

1. Compute the syndrome s=yHT for the received vector y. If s=0, then there are no errors. Return the received vector and exit.

2. Otherwise, determine the position j of the column of H that is the transpose of the syndrome.

3. Change the jth bit in the received word, and output

the resulting code.

As long as there is at most one bit error in the received vector, the result will be the codeword that was sent.

Example

The [15, 11] binary Hamming code has parity check matrix

$$0000111111111000111000011111010001110110011001010111010101000 1.$$

Assume the received vector is

$$y=(0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1).$$

The syndrome $s=yH^T$ is calculated to be $s=(1, 1, 1, 1)$. Notice that s is the transpose of the 11th column of H, so we change the 11th bit of y to get the decoded word as

(0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1).

Since the first 11 bits give the information, the original message was

(0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0).

Therefore, we have detected and corrected the error.

24.6 Golay Codes

Two of the most famous binary codes are the Golay codes G23 and G24. The [24, 12, 8] extended Golay code G24 was used by the *Voyager I* and *Voyager II* spacecrafts during 1979–1981 to provide error correction for transmission back to Earth of color pictures of Jupiter and Saturn. The (nonextended) Golay code G23, which is a [23, 12, 7] code, is closely related to G24. We shall construct G24 first, then modify it to obtain G23. There are many other ways to construct the Golay codes. See [MacWilliams-Sloane].

The generating matrix for G24 is the

12×24 matrix G=

100000000000111011100010010000000000101101110001001000000000110

All entries of G are integers mod 2. The first 12 columns of G are the 12×12 identity matrix. The last 11 columns are obtained as follows. The squares mod 11 are 0, 1, 3, 4, 5, 9 (for example, $42 \equiv 3$ and $72 \equiv 5$). Take the vector (x0, …, x10)=(1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0), with a 1 in positions 0, 1, 3, 4, 5, 9. This gives the last 11 entries in the first row of G. The last 11 elements of the other rows, except the last, are obtained by cyclically permuting the entries in this vector. (Note: The entries are integers mod 2, not mod 11. The squares mod 11 are used only to determine which positions receive a 1.) The 13th column

and the 12th row are included because they can be; they increase k and d and help give the code some of its nice properties. The basic properties of G24 are given in the following theorem.

Theorem

G24 is a self-dual [24, 12, 8] binary code. The weights of all vectors in G24 are multiples of 4.

Proof. The rows in G have length 24. Since the 12×12 identity matrix is contained in G, the 12 rows of G are linearly independent. Therefore, G24 has dimension 12, so it is a [24, 12, d] code for some d. The main work will be to show that d=8. Along the way, we'll show that G24 is self-dual and that the

weights of its codewords are 0 (mod4).

Of course, it would be possible to have a computer list all 212=4096 elements of G24 and their weights. We would then verify the claims of the theorem. However, we prefer to give a more theoretical proof.

Let r1 be the first row of G and let $r \neq r1$ be any of the other first 11 rows. An easy check shows that r1 and r have exactly four 1s in common, and each has four 1s that are matched with 0s in the other vector. In the sum r1+r, the four common 1s cancel mod 2, and the remaining four 1s from each row give a total of eight 1s in the sum. Therefore, r1+r has weight 8. Also, the dot product $r1 \cdot r$ receives contributions only from the

common 1s, so
$r1 \cdot r = 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 = 4 \equiv 0 \pmod 2$.

Now let u and v be any two distinct rows of G, other than the last row. The first 12 entries and the last 11 entries of v are cyclic permutations of the corresponding parts of u and also of the corresponding parts of the first row. Since a permutation of the entries does not change the weights of vectors or the value of dot products, the preceding calculation of r1+r and r1·r applies to u and v. Therefore,

1. $wt(u+v)=8$
2. $u \cdot v \equiv 0 \pmod 2$.

Any easy check shows that (1) and (2) also hold if u or v is the last row of G,

so we see that (1) and (2) hold for any two distinct rows u, v of G. Also, each row of G has an even number of 1s, so (2) holds even when u=v.

Now let c1 and c2 be arbitrary elements of G24. Then c1 and c2 are linear combinations of rows of G, so c1·c2 is a linear combination of numbers of the form u·v for various rows u and v of G. Each of these dot products is 0 mod 2, so $r1 \cdot r2 \equiv 0 \pmod 2$. This implies that $C \subseteq C^{\perp}$. Since C is a 12-dimensional subspace of 24-dimensional space, $C^{\perp}$ has dimension 24−12=12. Therefore, C and $C^{\perp}$ have the same dimension, and one is contained in the other. Therefore, $C = C^{\perp}$, which says that C is self-dual.

Observe that the weight of each row of

G is a multiple of 4. The following lemma will be used to show that every element of G24 has a weight that is a multiple of 4.

Lemma

Let v1 and v2 be binary vectors of the same length. Then

$$wt(v1+v2)=wt(v1)+wt(v2)-2[v1\cdot v2],$$

where the notation [v1·v2] means that the dot product is regarded as a usual integer, not mod 2 (for example, [(1, 0, 1, 1)·(1, 1, 1, 1)]=3, rather than 1).

Proof. The nonzero entries of v1+v2 occur when exactly one of the vectors v1, v2 has an entry 1 and the other has a

0 as its corresponding entry. When both vectors have a 1, these numbers add to 0 mod 2 in the sum. Note that wt(v1)+wt(v2) counts the total number of 1s in v1 and v2 and therefore includes these 1s that canceled each other. The contributions to [v1·v2] are caused exactly by these 1s that are common to the two vectors. So there are [v1·v2] entries in v1 and the same number in v2 that are included in wt(v1)+wt(v2), but do not contribute to wt(v1+v2). Putting everything together yields the equation in the lemma.

We now return to the proof of the theorem. Consider a vector g in G24. It can be written as a sum $g \equiv u1+\cdots+uk \pmod 2$, where u1, …, uk are distinct

rows of G. We'll prove that $wt(g) \equiv 0 \pmod 4$ by induction on k. Looking at G, we see that the weights of all rows of G are multiples of 4, so the case k=1 is true. Suppose, by induction, that all vectors that can be expressed as a sum of $k-1$ rows of G have weight $\equiv 0 \pmod 4$. In particular, $u=u1+\cdots+uk-1$ has weight a multiple of 4. By the lemma,

$$wt(g)=wt(u+uk)=wt(u)+wt(uk)-2[u\cdot uk] \equiv 0+0-2[u\cdot uk] \pmod 4.$$

But $u\cdot uk \equiv 0 \pmod 2$, as we proved. Therefore, $2[u\cdot uk] \equiv 0 \pmod 4$. We have proved that $wt(g) \equiv 0 \pmod 4$ whenever g is a sum of k rows. By induction, all sums of rows of G have weight $\equiv 0 \pmod 4$. This proves that all weights of G24 are multiples of 4.

Finally, we prove that the minimum weight in G24 is 8. This is true for the rows of G, but we also must show it for sums of rows of G. Since the weights of codewords are multiples of 4, we must show that there is no codeword of weight 4, since the weights must then be at least 8. In fact, 8 is then the minimum, because the first row of G, for example, has weight 8.

We need the following lemma.

Lemma

The rows of the 12×12 matrix B formed from the last 12 columns of G are linearly independent mod 2. The rows of the 11×11 matrix A formed from the last 11 elements of the first 11 rows of G are

linearly dependent mod 2. The only linear dependence relation is that the sum of all 11 rows of A is 0 mod 2.

Proof. Since G24 is self-dual, the dot product of any two rows of G is 0. This means that the matrix product GGT=0. Since G=[I|B] (that is, I followed by the matrix B), this may be rewritten as

I2+B BT=0,

which implies that B−1=BT (we're working mod 2, so the minus signs disappear). This means that B is invertible, so the rows are linearly independent.

The sum of the rows of A is 0 mod 2, so this is a dependence relation. Let v1=(1, …, 1)T be an 11-dimensional

column vector. Then Av1=0, which is just another way of saying that the sum of the rows is 0. Suppose v2 is a nonzero 11-dimensional column vector such that Av2=0. Extend v1 and v2 to 12-dimensional vectors v1′, v2′ by adjoining a 0 at the top of each column vector. Let r12 be the bottom row of B. Then

$$Bvi'=(0, \ldots, 0, r12\cdot vi')T.$$

This equation follows from the fact that Avi=0. Note that multiplying a matrix times a vector consists of taking the dot products of the rows of the matrix with the vector.

Since B is invertible and $vi' \neq 0$, we have $Bvi' \neq 0$, so $r1\cdot vi' \neq 0$ Since we are working mod 2, the dot product must

equal 1. Therefore,

$$B(v1'+v2')=(0, \ldots, 0, r1\cdot v1'+r1\cdot v2')T=(0, \ldots, 0, 1+1)T=0.$$

Since B is invertible, we must have $v1'+v2'=0$, so $v1'=v2'$ (we are working mod 2). Ignoring the top entries in $v1'$ and $v2'$, we obtain $v2=(1, \ldots, 1)$. Therefore, the only nonzero vector in the null space of A is v1. Since the vectors in the null space of a matrix give the linear dependencies among the rows of the matrix, we conclude that the only dependency among the rows of A is that the sum of the rows is 0. This proves the lemma.

Suppose g is a codeword in G24. If g is, for example, the sum of the second, third, and seventh rows, then g will have 1s in the second, third, and seventh

positions, because the first 12 columns of G form an identity matrix. In this way, we see that if g is the sum of k rows of G, then wt(g) ≥ k. Suppose now that wt(g)=4. Then g is the sum of at most four rows of G. Clearly, g cannot be a single row of G, since each row has weight at least 8. If g is the sum of two rows, we proved that wt(g) is 8. If g=r1+r2+r3 is the sum of three rows of G, then there are two possibilities.

(1) First, suppose that the last row of G is not one of the rows in the sum. Then three 1s are used from the 13th column, so a 1 appears in the 13th position of g. The 1s from the first 12 positions (one for each of the rows r1, r2, r3) contribute three more 1s to g. Since

wt(g)=4, we have accounted for all four 1s in g. Therefore, the last 11 entries of g are 0. By the preceding lemma, a sum of only three rows of the matrix A cannot be 0. Therefore, this case is impossible.

(2) Second, suppose that the last row of G appears in the sum for g, say g=r1+r2+r3 with r3= the last row of G. Then the last 11 entries of g are formed from the sum of two rows of A (from r1 and r2) plus the vector (1, 1, …, 1) from r3. Recall that the weight of the sum of two distinct rows of G is 8. There is a contribution of 2 to this weight from the first 13 columns. Therefore, looking at the last 11 columns, we see that the sum of two distinct rows of A has weight 6.

Adding a vector mod 2 to the vector (1, 1, …, 1) changes all the 1s to 0s and all the 0s to 1s. Therefore, the weight of the last 11 entries of g is 5. Since wt(g)=4, this is impossible, so this case also cannot occur.

Finally, if g is the sum of four rows of G, then the first 12 entries of g have four 1s. Therefore, the last 12 entries of g are all 0. By the lemma, a sum of four rows of B cannot be 0, so we have a contradiction. This completes the proof that there is no codeword of weight 4.

Since the weights are multiples of 4, the smallest possibility for the weight is 8. As we pointed out previously, there are codewords of weight 8, so we have proved that the minimum weight of G24

is 8. Therefore, G24 is a [24, 12, 8] code, as claimed. This completes the proof of the theorem.

The (nonextended) Golay code G23 is obtained by deleting the last entry of each codeword in G24.

Theorem

G23 is a linear [23,12,7] code.

Proof. Clearly each codeword has length 23. Also, the set of vectors in G23 is easily seen to be closed under addition (if v1, v2 are vectors of length 24, then the first 23 entries of v1+v2 are computed from the first 23 entries of v1 and v2) and G23 forms a binary vector space. The generating matrix G′ for G23

is obtained by removing the last column of the matrix G for G24. Since G′ contains the 12×12 identity matrix, the rows of G′ are linearly independent, and hence span a 12-dimensional vector space. If g′ is a codeword in G23, then g′ can be obtained by removing the last entry of some element g of G24. If g′ ≠ 0, then g ≠ 0, so wt(g) ≥ 8. Since g′ has one entry fewer than g, we have wt(g′) ≥ 7. This completes the proof.

Decoding G24

Suppose a message is encoded using G24 and the received message contains at most three errors. In the following, we show a way to correct these errors.

Let G be the 12×24 generating matrix

for G24. Write G in the form

$$G=[I, B]=(c1, \cdots, c24),$$

where I is the 12×12 identity matrix, B consists of the last 12 columns of G, and c1, …, c24 are column vectors. Note that c1, …, c12 are the standard basis elements for 12-dimensional space. Write

$$BT=(b1, \ldots, b12),$$

where b1, …, b12 are column vectors. This means that b1T, …, b12T are the rows of B.

Suppose the received message is r=c+e, where c is a codeword from G24 and

$$e=(e1, \ldots, e24)$$

is the error vector. We assume wt(e) ≤ 3.

The algorithm is as follows. The justification is given below.

1. Let s=rGT be the syndrome.

2. Compute the row vectors s, sB, s+cjT, $13 \le j \le 24$, and sB+bjT, $1 \le j \le 12$.

3. If $wt(s) \le 3$, then the nonzero entries of s correspond to the nonzero entries of e.

4. If $wt(sB) \le 3$, then there is a nonzero entry in the kth position of sB exactly when the (k+12)th entry of e is nonzero.

5. If $wt(s+cjT) \le 2$ for some j with $13 \le j \le 24$, then ej=1 and the nonzero entries of s+cjT are in the positions of the other nonzero entries of the error vector e.

6. If $wt(sB+bjT) \le 2$ for some j with $1 \le j \le 12$, then ej=1. If there is a nonzero entry for this sB+bjT in position k (there are at most two such k), then e12+k=1.

Example

The sender starts with the message

m=(1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0).

The codeword is computed as

mG=(1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0)

and sent to us. Suppose we receive the message as

r=(1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0).

A calculation shows that

s=(0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0)

and

sB=(1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0).

Neither of these has weight at most 3, so we compute s+cjT, $13 \leq j \leq 24$ and sB +bjT, $1 \leq j \leq 12$. We find that

sB+b4T=(0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0).

This means that there is an error in position 4 (corresponding to the choice b4) and in positions 20 (= 12 + 8) and 22 (= 12 + 10) (corresponding to the nonzero entries in positions 8 and 10 of sB+b4T). We therefore compute

$$c=r+(0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0)=(1,1,0,0,0,0,$$

Moreover, since G is in systematic form, we recover the original message from the first 12 entries:

$$m=(1,1,0,0,0,0,0,0,0,0,1,0).$$

We now justify the algorithm and show that if $wt(e) \leq 3$, then at least one of the preceding cases occurs.

Since G24 is self-dual, the dot product of a row of G with any codeword c is 0.

This means that cGT=0. In our case, we have r=c+e, so

s=rGT=cGT+eGT=eGT=e1c1T+⋯+e24c24T.

This last equality just expresses the fact that the vector e=(e1, …, e24) times the matrix GT equals e1 times the first row c1T of GT, plus e2 times the second row of GT, etc. Also,

sB=eGTB=e[IBT]B=e[BI],

since BT=B−1 (proved in the preceding lemma). We have

BI=[BT, I]T=(b1, …, b12, c1, …, c12).

Therefore,

sB=e(b1, …, b12, c1, …, c12)T=e1b1T+…+e24c12T.

If wt(e) ≤ 3, then either
wt((e1, …, e12)) ≤ 1 or

wt((e13, …, e24)) ≤ 1, since otherwise there would be too many nonzero entries in e. We therefore consider the following four cases.

1. wt((e1, …, e12))=0. Then

   sB=e13c1T+⋯+e24c12T=(e13, …, e24).

   Therefore, wt(sB) ≤ 3 and we can determine the errors as in step (4) of the algorithm.

2. wt((e1, …, e12))=1. Then ej=1 for exactly one j with 1 ≤ j ≤ 12, so

   sB=bjT+e13c1T+⋯+e24c12T.

   Therefore,

   sB+bjT=e13c1T+…+e24c12T=(e13, …, e24).

   The vector (e13, …, e24) has at most two nonzero entries, so we are in step (6) of the algorithm.

   The choice of j is uniquely determined by sB. Suppose wt(sB+bkT) ≤ 2 for some k ≠ j. Then

   wt(bkT+bjT)=wt(sB+bkT+sB+bjT) ≤ wt(sB +bkT)+wt(sB+bjT) ≤ 2+2=4

(see Exercise 6). However, we showed in the proof of the theorem about G24 that the weight of the sum of any two distinct rows of G has weight 8, from which it follows that the sum of any two distinct rows of B has weight 6. Therefore, wt(bkT +bjT)=6. This contradiction shows that bk cannot exist, so bj is unique.

3. wt((e13, …, e24))=0. In this case,

   s=e1c1T+⋯+e12c12T=(e1, …, e12).

   We have wt(s) ≤ 3, so we are in step (3) of the algorithm.

4. wt((e13, …, e24))=1. In this case, ej=1 for some j with 13 ≤ j ≤ 24. Therefore,

   s=e1c1T+⋯+e12c12T+cjT,

   and we obtain

   s+cjT=e1c1T+⋯+e12c12T=(e1, …, e12).

   There are at most two nonzero entries in (e1, …, e12), so we are in step (5) of the algorithm.

   As in (2), the choice of cj is uniquely determined by s.

In each of these cases, we obtain a

vector, let's call it e′, with at most three nonzero entries. To correct the errors, we add (or subtract; we are working mod 2) e′ to the received vector r to get c′=r+e′. How do we know this is the vector that was sent? By the choice of e ′, we have

e′GT=s,

so

c′GT=rGT+e′GT=s+s=0.

Since G24 is self-dual, G is a parity check matrix for G24. Since c′GT=0, we conclude that c′ is a codeword. We obtained c′ by correcting at most three errors in r. Since we assumed there were at most three errors, and since the minimum weight of G24 is 8, this must be the correct decoding. So the

algorithm actually corrects the errors, as claimed.

The preceding algorithm requires several steps. We need to compute the weights of 26 vectors. Why not just look at the various possibilities for three errors and see which correction yields a codeword? There are $240+241+242+243=2325$ possibilities for the locations of at most three errors, so this could be done on a computer. However, the preceding decoding algorithm is faster.

24.7 Cyclic Codes

Cyclic codes are a very important class of codes. In the next two sections, we'll meet two of the most useful examples of these codes. In this section, we describe the general framework.

A code C is called **cyclic** if

$(c1, c2, \ldots, cn) \in C$ implies $(cn, c1, c2, \ldots, cn-1) \in C$.

For example, if (1, 1, 0, 1) is in a cyclic code, then so is (1, 1, 1, 0). Applying the definition two more times, we see that (0, 1, 1, 1) and (1, 0, 1, 1) are also codewords, so all cyclic permutations of the codeword are codewords. This might seem to be a strange condition for a

code to satisfy. After all, it would seem to be rather irrelevant that, for a given codeword, all of its cyclic shifts are still codewords. The point is that cyclic codes have a lot of structure, which makes them easier to study. In the case of BCH codes (see Section 24.8), this structure yields an efficient decoding algorithm.

Let's start with an example. Consider the binary matrix

$$G=\begin{matrix}1011\\0001\\0001\end{matrix}\begin{matrix}0001\\1100\\0111\end{matrix}.$$

The rows of G generate a three-dimensional subspace of seven-dimensional binary space. In fact, in this case, the cyclic shifts of the first row give all the nonzero codewords:

G={(0, 0, 0, 0, 0, 0, 0), (1, 0, 1, 1, 1, 0, 0), (0, 1, 0, 1, 1, 1, 0), (0, 0, 1, 0, 1, 1, 1

Clearly the minimum weight is 4, so we have a cyclic [7, 3, 4] code.

We now show an algebraic way to obtain this code. Let Z2[X] denote polynomials in X with coefficients mod 2, and let Z2[X]/(X7−1) denote these polynomials mod (X7−1). For a detailed description of what this means, see Section 3.11. For the present, it suffices to say that working mod X7−1 means we are working with polynomials of degree less than 7. Whenever we have a polynomial of degree 7 or higher, we divide by X7−1 and take the remainder.

Let g(X)=1+X2+X3+X4. Consider all products

g(X)f(X)=a0+a1X+⋯+a6X6

with f(X) of degree ≤ 2. Write the coefficients of the product as a vector (a0, …, a6). For example, g(X)·1 yields (1, 0, 1, 1, 1, 0, 0), which is the top row of G. Similarly, g(X)X yields the second row of G and g(X)X2 yields the third row of G. Also, g(X)(1+X2) yields (1, 0, 0, 1, 0, 1, 1), which is the sum of the first and third rows of G. In this way, we obtain all the codewords of our code.

We obtained this code by considering products g(X)f(X) with deg(f) ≤ 2. We could also work with f(X) of arbitrary degree and obtain the same code, as long as we work mod (X7−1). Note that g(X)(X3+X2+1)=X7−1 (mod 2). Divide X3+X2+1 into f(X):

$$f(X)=(X3+X2+1)q(X)+f1(X),$$

with deg(f1) ≤ 2. Then

$$g(X)f(X)=g(X)(X3+X2+1)q(X)+g(X)f1(X)=(X7−1)q(X)+g(X)f1(X)\equiv g(X)f1(X) \mod (X7−1).$$

Therefore, g(X)f1(X) gives the same codeword as g(X)f(X), so we may restrict to working with polynomials of degree at most two, as claimed.

Why is the code cyclic? Start with the vector for g(X). The vectors for g(X)X and g(X)X2 are cyclic shifts of the one for g(X) by one place and by two places, respectively. What happens if we multiply by X3? We obtain a polynomial of degree 7, so we divide by X7−1 and take the remainder:

$$g(X)X3=X3+X5+X6+X7=(X7−1)(1)+(1+X3+X5+X6).$$

The remainder yields the vector (1, 0, 0, 1, 0, 1, 1). This is the cyclic shift by three places of the vector for g(X).

A similar calculation for j=4, 5, 6 shows that the vector for g(X)Xj yields the shift by j places of the vector for g(X). In fact, this is a general phenomenon. If q(X)=a0+a1X+⋯+a6X6 is a polynomial, then

q(X)X=a0X+a1X2+…+a6X7=a6(X7−1)+a6+a0X+a1X2+⋯ +a5X6.

The remainder is a6+a0X+a1X2+⋯ +a5X6, which corresponds to the vector (a6, a0, …, a5). Therefore, multiplying by X and reducing mod X7−1 corresponds to a cyclic shift by one place of the corresponding vector.

Repeating this j times shows that multiplying by Xj corresponds to shifting by j places.

We now describe the general situation. Let F be a finite field. For a treatment of finite fields, see Section 3.11. For the present purposes, you may think of F as being the integers mod p, where p is a prime number, since this is an example of a finite field. For example, you could take F=Z2={0, 1}, the integers mod 2. Let F[X] denote polynomials in X with coefficients in F. Choose a positive integer n. We'll work in F[X]/(Xn−1), which denotes the elements of F[X] mod (Xn−1). This means we're working with polynomials of degree less than n. Whenever we encounter a polynomial of

degree $\geq n$, we divide by X^n-1 and take the remainder. Let $g(X)$ be a polynomial in $F[X]$. Consider the set of polynomials

$$m(X)=g(X)f(X) \bmod (X^n-1),$$

where $f(X)$ runs through all polynomials in $F[X]$ (we only need to consider $f(X)$ with degree less than n, since higher-degree polynomials can be reduced mod X^n-1). Write

$$m(X)=a_0+a_1X+\cdots+a_{n-1}X^{n-1}.$$

The coefficients give us the n-dimensional vector $(a_0, \ldots, a_{n-1})$. The set of all such coefficients forms a subspace C of n-dimensional space F^n. Then C is a code.

If $m(X)=g(X)f(X) \bmod (X^n-1)$ is any such polynomial, and $s(X)$ is another

polynomial, then $m(X)s(X)=g(X)f(X)s(X) \bmod (X^n-1)$ is the multiple of $g(X)$ by the polynomial $f(X)s(X)$. Therefore, it yields an element of the code C. In particular, multiplication by X and reducing mod X^n-1 corresponds to a codeword that is a cyclic shift of the original codeword, as above. Therefore, C is cyclic.

The following theorem gives the general description of cyclic codes.

Theorem

Let C be a cyclic code of length n over a finite field F. To each codeword $(a_0, \ldots, a_{n-1}) \in C$, associate the polynomial $a_0+a_1X+\cdots+a_{n-1}X^{n-1}$ in $F[X]$. Among all the nonzero

polynomials obtained from C in this way, let g(X) have the smallest degree. By dividing by its highest coefficient, we may assume that the highest nonzero coefficient of g(X) is 1. The polynomial g(X) is called the **generating polynomial** for C. Then

1. g(X) is uniquely determined by C.
2. g(X) is a divisor of X^n-1.
3. C is exactly the set of coefficients of the polynomials of the form g(X)f(X) with $\deg(f) \leq n-1-\deg(g)$.
4. Write $X^n-1=g(X)h(X)$. Then $m(X) \in F[X]/(X^n-1)$ corresponds to an element of C if and only if $h(X)m(X) \equiv 0 \bmod (X^n-1)$.

Proof.

1. If g1(X) is another such polynomial, then g(X) and g1(X) have the same degree and have highest nonzero coefficient equal to 1. Therefore,

g(X)−g1(X) has lower degree and still corresponds to a codeword, since C is closed under subtraction. Since g(X) had the smallest degree among nonzero polynomials corresponding to codewords, g(X)−g1(X) must be 0, which means that g1(X)=g(X). Therefore, g(X) is unique.

2. Divide g(X) into Xn−1:

   Xn−1=g(X)h(X)+r(X)

   for some polynomials h(X) and r(X), with deg(r)tdeg(g). This means that

   −r(X) ≡ g(X)h(X) mod (Xn−1).

   As explained previously, multiplying g(X) by powers of X corresponds to cyclic shifts of the codeword associated to g(X). Since C is assumed to be cyclic, the polynomials g(X)Xj mod (Xn−1) for j=0, 1, 2, … therefore correspond to codewords; call them c0, c1, c2, …. Write h(X)=b0+b1X+⋯+bkXk. Then g(X)h(X) corresponds to the linear combination

   b0c0+b1c1+⋯+bkck.

   Since each bi is in F and each ci is in C, we have a linear combination of elements of C. But C is a vector subspace of n-dimensional space Fn. Therefore, this linear combination is in C. This means that r(X), which is g(X)h(X) mod (Xn−1),

corresponds to a codeword. But deg(r)tdeg(g), which is the minimal degree of a polynomial corresponding to a nonzero codeword in C. Therefore, r(X)=0. Consequently Xn−1=g(X)h(X), so g(X) is a divisor of Xn−1.

3. Let m(X) correspond to an element of C. Divide g(X) into m(X):

   m(X)=g(X)f(X)+r1(X),

   with deg(r1(X))<deg(g(X)). As before, g(X)f(X) mod (Xn−1) corresponds to a codeword. Also, m(X) corresponds to a codeword, by assumption. Therefore, m(X)−g(X)f(X) mod (Xn−1) corresponds to the difference of these codewords, which is a codeword. But this polynomial is just r1(X)=r1(X) mod (Xn−1). As before, this polynomial has degree less than deg(g(X)), so r1(X)=0. Therefore, m(X)=g(X)f(X). Since deg(m) ≤ n−1, we must have deg((f) ≤ n−1−deg(g). Conversely, as explained in the proof of (2), since C is cyclic, any such polynomial of the form g(X)f(X) yields a codeword. Therefore, these polynomials yield exactly the elements of C.

4. Write Xn−1=g(X)h(X), which can be done by (2). Suppose m(X) corresponds to an element of C. Then m(X)=g(X)f(X), by (3), so

   h(X)m(X)=h(X)g(X)f(X)=(Xn−1)f(X) ≡ 0 mod (Xn

−1).

Conversely, suppose m(X) is a polynomial such that $h(X)m(X) \equiv 0 \mod (X^n-1)$. Write $h(X)m(X)=(X^n-1)q(X)=h(X)g(X)q(X)$, for some polynomial q(X). Dividing by h(X) yields $m(X)=g(X)q(X)$, which is a multiple of g(X), and hence corresponds to a codeword. This completes the proof of the theorem.

Let $g(X)=a_0+a_1X+\cdots+a_{k-1}X^{k-1}+X^k$ be as in the theorem. By part (3) of the theorem, every element of C corresponds to a polynomial of the form g(X)f(X), with $\deg(f(X)) \le n-1-k$. This means that each such f(X) is a linear combination of the monomials $1, X, X^2, \ldots, X^{n-1-k}$. It follows that the codewords of C are linear combinations of the codewords corresponding to the polynomials

$g(X),\ g(X)X,\ g(X)X^2, \ldots,\ g(X)X^{n-1-k}.$

But these are the vectors

a0, … , ak, 0, 0, …), (0, a0, … , ak, 0, …), … , (0, … , 0, a0 , … , ak).

Therefore, a generating matrix for C can be given by

G=a0a1⋯ak00⋯0a0a1⋯ak0⋯⋮⋮⋮⋮0⋯0a0a1⋯ak.

We can use part (4) of the theorem to obtain a parity check matrix for C. Let h(X)=b0+b1X+⋯+blXl be as in the theorem (where l=n−k). We'll prove that the k×n matrix

H=blbl−1⋯b000⋯0blbl−1⋯b00⋯⋮⋮⋮⋮0⋯0blbl−1⋯b0

is a parity check matrix for C. Note that the order of the coefficients of h(X) is reversed. Recall that H is a parity check matrix for C means that HcT=0 if and only if c ∈ C.

Proposition

H is a parity check matrix for C.

Proof. First observe that since g(X) has 1 as its highest nonzero coefficient, and since g(X)h(X)=Xn−1, the highest nonzero coefficient bl of h(X) must also be 1. Therefore, H is in row echelon form and consequently its rows are linearly independent. Since H has k rows, it has rank k. The right null space of H therefore has dimension n−k.

Let m(X)=c0+c1X+⋯+cn−1Xn−1. We know from part (4) that (c0, c1, …, cn −1) ∈ C if and only if h(X)m(X) ≡ 0 mod (Xn−1).

Choose j with l ≤ j ≤ n−1 and look at the

coefficient of Xj in the product h(X)m(X). It equals

b0cj+b1cj−1+⋯+bl−1cj−l+1+blcj−l.

There is a technical point to mention: Since we are looking at h(X)m(X) mod (Xn−1), we need to worry about a contribution from the term Xn+j (since Xn+j ≡ XnXj ≡ 1·Xj, the monomial Xn +j reduces to Xj). However, the highest-degree term in the product h(X)m(X) before reducing mod Xn−1 is cn−1Xl+n −1. Since l ≤ j, we have l+n−1tj+n. Therefore, there is no term with Xn+j to worry about.

When we multiply H times (c0, c1, …, cn−1)T, we obtain a vector whose first entry is

blc0+bl−1c1+⋯+b0cl.

More generally, the ith entry (where $1 \le i \le k$) is

blci−1+bl−1ci+⋯+b0cl+i−1.

This is the coefficient of Xl+i−1 in the product h(X)m(X) mod (Xn−1).

If (c0, c1, …, cn−1) is in C, then $h(X)m(X) \equiv 0 \bmod (X^n-1)$, so all these coefficients are 0. Therefore, H times (c0, c1, …, cn−1)T is the 0 vector, so the transposes of the vectors of C are contained in the right null space of H. Since both C and the null space have dimension k, we must have equality. This proves that $c \in C$ if and only if HcT=0, which means that H is a parity check matrix for C.

Example

In the example at the beginning of this section, we had n=7 and g(X)=X4+X3+X2+1. We have g(X)(X3+X2+1)=X7−1, so h(X)=X3+X2+1. The parity check matrix is

H=11010000110100001101000011101.

The parity check matrix gives a way of detecting errors, but correcting errors for general cyclic codes is generally quite difficult. In the next section, we describe a class of cyclic codes for which a good decoding algorithm exists.

24.8 BCH Codes

BCH codes are a class of cyclic codes. They were discovered around 1959 by R. C. Bose and D. K. Ray-Chaudhuri and independently by A. Hocquenghem. One reason they are important is that there exist good decoding algorithms that correct multiple errors (see, for example, [Gallager] or [Wicker]). BCH codes are used in satellites. The special BCH codes called Reed-Solomon codes (see Section 24.9) have numerous applications.

Before describing BCH codes, we need a fact about finite fields. Let F be a finite field with q elements. From Section

3.11, we know that $q=p^m$ is a power of a prime number p. Let n be a positive integer not divisible by p. Then it can be proved that there exists a finite field F' containing F such that F' contains a primitive nth root of unity α. This means that $\alpha^n=1$, but $\alpha^k \neq 1$ for $1 \leq k<n$.

For example, if $F=Z_2$, the integers mod 2, and $n=3$, we may take $F'=GF(4)$. The element ω in the description of $GF(4)$ in Section 3.11 is a primitive third root of unity. More generally, a primitive nth root of unity exists in a finite field F' with q' elements if and only if $n|q'-1$.

The reason we need the auxiliary field F' is that several of the calculations we perform need to be carried out in this larger field. In the following, when we

use an nth root of unity α, we'll implicitly assume that we're calculating in some field F' that contains α. The results of the calculations, however, will give results about codes over the smaller field F.

The following result, often called the **BCH bound**, gives an estimate for the minimum weight of a cyclic code.

Theorem 24.8

Let C be a cyclic [n, k, d] code over a finite field F, where F has $q=p^m$ elements. Assume $p|n$. Let $g(X)$ be the generating polynomial for C. Let α be a primitive nth root of unity and suppose that for some integers ℓ and δ,

$$g(\alpha^{\ell})=g(\alpha^{\ell+1})=\cdots=g(\alpha^{\ell+\delta})=0.$$

Then $d \geq \delta+2$.

Proof. Suppose $(c0, c1, \ldots, cn-1) \in C$ has weight w with $1 \leq w<\delta+2$. We want to obtain a contradiction. Let $m(X)=c0+c1X+\cdots+cn-1Xn-1$. We know that m(X) is a multiple of g(X), so

$$m(\alpha\ell)=m(\alpha\ell+1)=\cdots=m(\alpha\ell+\delta)=0.$$

Let ci1, ci2, …, ciw be the nonzero coefficients of m(X), so

$$m(X)=ci1Xi1+ci2Xi2+\cdots+ciwXiw.$$

The fact that $m(\alpha j)=0$ for $l \leq j \leq \ell+w-1$ (note that $w-1 \leq \delta$) can be rewritten as

$$(\alpha\ell i1\cdots\alpha\ell iw\alpha(\ell+1)i1\cdots\alpha(\ell+1)iw\vdots\ddots\vdots\alpha(\ell+w-1)i1\cdots\alpha(\ell+w-1)iw)(ci1ci2\vdots ciw)=(00\vdots 0).$$

We claim that the determinant of the matrix is nonzero. We need the

following evaluation of the Vandermonde determinant. The proof can be found in most books on linear algebra.

Proposition

det11⋯1x1x2⋯xnx12x22⋯xn2⋮⋮⋱⋮x1n−1x2n−1⋯xnn−1=∏1 ≤ itj ≤ n(xj−xi).

(The product is over all pairs of integers (i, j) with 1 ≤ i<j ≤ n.) In particular, if x1, …, xn are pairwise distinct, the determinant is nonzero.

In our matrix, we can factor αℓi1 from the first column, αℓi2 from the second column, etc., to obtain

det(αℓi1⋯αℓiwα(ℓ+1)i1⋯α(ℓ+1)iw⋮⋱⋮α(ℓ+w−1)i1⋯α(ℓ+w−1)iw)=αℓi1+⋯+ℓiwdet(1⋯1αi1⋯αiw⋮⋱⋮α(w−1)i1⋯α(w−1)iw).

Since αi1, ⋯, αiw are pairwise distinct,

the determinant is nonzero. Why are these numbers distinct? Suppose $\alpha ij=\alpha ik$. We may assume $ij \leq ik$. We have $0 \leq ij \leq ik<n$. Therefore, $0 \leq ik-ij<n$. Note that $\alpha ik-ij=1$. Since α is a primitive nth root of unity, $\alpha i \neq 1$ for $1 \leq i<n$. Therefore, $ik-ij=0$, so $ij=ik$. This means that the numbers $\alpha i1, \cdots, \alpha iw$ are pairwise distinct, as claimed.

Since the determinant is nonzero, the matrix is nonsingular. This implies that $(ci1, \ldots, ciw)=0$, contradicting the fact that these were the nonzero ci's. Therefore, all nonzero codewords have weight at least $\delta+2$. This completes the proof of the theorem.

Example

Let F=Z2= the integers mod 2, and let n=3. Let g(X)=X2+X+1. Then

C={(0, 0, 0), (1, 1, 1)},

which is a binary repetition code. Let ω be a primitive third root of unity, as in the description of GF(4) in Section 3.11. Then g(ω)=g(ω2)=0. In the theorem, we can therefore take ℓ=1 and δ=1. We find that the minimal weight of C is at least 3. In this case, the bound is sharp, since the minimal weight of C is exactly 3.

Example

Let F be any finite field and let n be any positive integer. Let g(X)=X−1. Then g(1)=0, so we may take ℓ=0 and δ=0.

We conclude that the minimum weight of the code generated by g(X) is at least 2 (actually, the theorem assumes that p| n, but this assumption is not needed for this special case where $\ell=\delta=0$). We have seen this code before. If (c0, …, cn−1) is a vector, and $m(X)=c0+\cdots+cn-1Xn-1$ is the associated polynomial, then m(X) is a multiple of X−1 exactly when m(1)=0. This means that $c0+\cdots+cn-1=0$. So a vector is a codeword if and only if the sum of its entries is 0. When F=Z2, this is the parity check code, and for other finite fields it is a generalization of the parity check code. The fact that its minimal weight is 2 is easy to see directly: If a codeword has a nonzero entry, then it must contain another nonzero entry to cancel it and make the

sum of the entries be 0. Therefore, each nonzero codeword has at least two nonzero entries, and hence has weight at least 2. The vector (1, −1, 0, …) is a codeword and has weight 2, so the minimal weight is exactly 2.

Example

Let’s return to the example of a binary cyclic code of length 7 from Section 24.7. We have F=Z2, and g(X)=1+X2+X3+X4. We can factor g(X)=(X−1)(X3+X+1). Let α be a root of X3+X+1. Then α is a primitive seventh root of unity (see Exercise 18), and we are working in GF(8). Since Z2 ⊂ GF(8), we have 2=1+1=0 and −1=1. Therefore, α3=α+1. Squaring yields α6=α2+2α+1=α2+1. Therefore,

$(\alpha 2)3+(\alpha 2)+1=0$. This means that $g(\alpha 2)=0$, so

$$g(1)=g(\alpha)=g(\alpha 2)=0.$$

In the theorem, we can take $\ell=0$ and $\delta=2$. Therefore, the minimal weight in the code is at least 4 (in fact, it is exactly 4).

To define the BCH codes, we need some more notation. We are going to construct codes of length n over a finite field F. Factor $Xn-1$ into irreducible factors over F:

$$Xn-1=f1(X)f2(X)\cdots fr(X),$$

where each fi(X) is a polynomial with coefficients in F, and each fi(X) cannot be factored into lower-degree polynomials with coefficients in F. We

may assume that the highest nonzero coefficient of each $f_i(X)$ is 1. Let α be a primitive nth root of unity. Then $\alpha 0, \alpha 1, \alpha 2, \ldots, \alpha n-1$ are roots of $Xn-1$. This means that

$$Xn-1=(X-1)(X-\alpha)(X-\alpha 2)\cdots(X-\alpha n-1).$$

Therefore, each $f_i(X)$ is a product of some of these factors $(X-\alpha j)$, and each αj is a root of exactly one of the polynomials $f_i(X)$. For each j, let $q_j(X)$ be the polynomial $f_i(X)$ such that $f_i(\alpha j)=0$. This gives us polynomials $q0(X), q1(X), \ldots qn-1(X)$. Of course, usually these polynomials are not all distinct, since a polynomial $f_i(X)$ that has two different powers $\alpha j, \alpha k$ as roots will serve as both $q_j(X)$ and $q_k(X)$ (see the examples given later in this section).

A **BCH code of designed distance d** is a code with generating polynomial

$$g(X)=\text{least common multiple of } q_{k+1}(X), q_{k+2}(X), \ldots, q_{k+d-1}(X)$$

for some integer k.

Theorem

A BCH code of designed distance d has minimum weight greater than or equal to d.

Proof. Since $q_j(X)$ divides $g(X)$ for $k+1 \le j \le k+d-1$, and $q_j(\alpha^j)=0$, we have

$$g(\alpha^{k+1})=g(\alpha^{k+2})=\cdots=g(\alpha^{k+d-1})=0.$$

The BCH bound (with $\ell=k+1$ and $\delta=d-2$) implies that the code has minimum weight at least $d=\delta+2$.

Example

Let F=Z2, and let n=7. Then

$$X7−1=(X−1)(X3+X2+1)(X3+X+1).$$

Let α be a root of X3+X+1. Then α is a primitive 7th root of unity, as in the previous example. Moreover, in that example, we showed that α2 is also a root of X3+X+1. In fact, we actually showed that the square of a root of X3+X+1 is also a root, so we have that α4=(α2)2 is also a root of X3+X+1. (We could square this again, but α8=α, so we are back to where we started.) Therefore, α, α2, α4 are the roots of X3+X+1, so

$$X3+X+1=(X−α)(X−α2)(X−α4).$$

The remaining powers of α must be

roots of X3+X2+1, so

X3+X2+1=(X−α3)(X−α5)(X−α6).

Therefore,

q0(X)=X−1, q1(X)=q2(X)=q4(X)=X3+X
+1, q3(X)=q5(X)=q6(X)=X3+X2+1.

If we take k=−1 and d=3, then

g(X)=lcm(q0(X), q1(X))=(X−1)(X3+X+1)=X4+X3+X2+1.

We obtain the cyclic [7, 3, 4] code discussed in Section 24.7. The theorem says that the minimum weight is at least 3. In this case, we can do a little better. If we take k=−1 and d=4, then we have a generating polynomial g1(X) with

g1(X)=lcmq0(X), q1(X), q2(X)=g(X).

This is because q2(X)=q1(X), so the least common multiple doesn't change

when q2(X) is included. The theorem now tells us that the minimum weight of the code is at least 4. As we have seen before, the minimum weight is exactly 4.

Example (continued)

Let's continue with the previous example, but take k=0 and d=7. Then

$$g(X)=\text{lcm}(q1(X), \ldots, q6(X))=(X3+X+1)(X3+X2+1)=X6+X5+X4+X3+X2+X+1.$$

We obtain the repetition code with only two codewords:

$$\{(0, 0, 0, 0, 0, 0, 0), (1, 1, 1, 1, 1, 1, 1)\}.$$

The theorem says that the minimum distance is at least 7. In fact it is exactly 7.

Example

Let F=Z5={0, 1, 2, 3, 4} = the integers mod 5. Let n=4. Then

$$X4−1=(X−1)(X−2)(X−3)(X−4)$$

(this is an equality, or congruence if you prefer, in Z5). Let $\alpha=2$. We have $24=1$, but $2j \neq 1$ for $1 \leq j<4$. Therefore, 2 is a primitive 4th root of unity in Z5. We have $20=1$, $22=4$, $23=3$ (these are just congruences mod 5). Therefore,

$$q0(X)=X−1,\ q1(X)=X−2,\ q2(X)=X−4,\ q3(X)=X−3.$$

In the theorem, let k=0, d=3. Then

$$g(X)=lcm(q1(X), q2(X))=(X−2)(X−4)=X2−6X+8=X2+4X+3.$$

We obtain a cyclic [4, 2] code over Z5 with generating matrix

$$34100341.$$

The theorem says that the minimum weight is at least 3. Since the first row of the matrix is a codeword of weight 3, the minimum weight is exactly 3. This code is an example of a Reed-Solomon code, which will be discussed in the next section.

24.8.1 Decoding BCH Codes

One of the reason BCH codes are useful is that there are good decoding algorithms. One of the best known is due to Berlekamp and Massey (see [Gallager] or [Wicker]). In the following, we won't give the algorithm, but, in order to give the spirit of some of the ideas that are involved, we show a way to correct one error in a BCH code

with designed distance $d \geq 3$.

Let C be a BCH code of designed distance $d \geq 3$. Then C is a cyclic code, say of length n, with generating polynomial g(X). There is a primitive nth root of unity α such that

$$g(\alpha k+1)=g(\alpha k+2)=0$$

for some integer k.

Let

$$H=1\alpha k+1\alpha 2(k+1)\cdots\alpha(n-1)(k+1)1\alpha k+2\alpha 2(k+2)\cdots\alpha(n-1)(k+2).$$

If $c=(c0, \ldots, cn-1)$ is a codeword, then the polynomial $m(X)=c0+c1X+\cdots+cn-1Xn-1$ is a multiple of g(X), so

$$m(\alpha k+1)=m(\alpha k+2)=0.$$

This may be rewritten in terms of H:

$cHT=(c0,\ c1,\ \ldots,\ cn-1)11\alpha k+1\alpha k+2\alpha 2(k+1)\alpha 2(k+2)\vdots\alpha(n-1)(k+1)\alpha(n-1)(k+2)=0.$

H is not necessarily a parity check matrix for C, since there might be noncodewords that are also in the null space of H. However, as we shall see, H can correct an error.

Suppose the vector r=c+e is received, where c is a codeword and e=(e0, …, en −1) is an error vector. We assume that at most one entry of e is nonzero.

Here is the algorithm for correcting one error.

1. Write rHT=(s1, s2).
2. If s1=0, there is no error (or there is more than one error), so we're done.
3. If $s1 \neq 0$, compute s2/s1. This will be a power $\alpha j-1$ of α. The error is in the jth position. If we are

working over the finite field Z2, we are done, since then ej=1. But for other finite fields, there are several choices for the value of ej.

4. Compute ej=s1/α(j−1)(k+1). This is the jth entry of the error vector e. The other entries of e are 0.

5. Subtract the error vector e from the received vector r to obtain the correct codeword c.

Example

Let's look at the BCH code over Z2 of length 7 and designed distance 7 considered previously. It is the binary repetition code of length 7 and has two codewords:
(0, 0, 0, 0, 0, 0, 0), (1, 1, 1, 1, 1, 1, 1).
The algorithm corrects one error. Suppose the received vector is r=(1, 1, 1, 1, 0, 1, 1). As before, let α be a root of X3+X+1. Then α is a primitive 7th root of unity.

Before proceeding, we need to deduce a few facts about computing with powers of α. We have α3=α+1. Multiplying this relation by powers of α yields

α4=α2+α, α5=α3+α2=α2+α+1, α6=α3+α2+α=(α+1)+α2+α=α2+1.

Also, the fact that αj=αj (mod 7) is useful.

We now can compute

rHT=(1, 1, 1, 1, 0, 1, 1)(11αα2α2α4⋮α6α12)=(1+α
+α2+α3+α5+α6, 1+α2+α4+α6+α10+α12)=(α+α2, α).

The sum in the first entry, for example, can be evaluated as follows:

1+α+α2+α3+α5+α6=1+α+α2+(1+α)+(α2+α+1)+(α2+1)=α+α2.

Therefore, s1=α+α2 and s2=α. We need to calculate s2/s1. Since s1=α+α2=α4, we have

$$s2/s1=\alpha/\alpha 4=\alpha-3=\alpha 4.$$

Therefore, $j-1=4$, so the error is in position $j=5$. The fifth entry of the error vector is $s1/\alpha 4=1$, so the error vector is $(0, 0, 0, 0, 1, 0, 0)$. The corrected message is

$$r-e=(1, 1, 1, 1, 1, 1, 1).$$

Here is why the algorithm works. Since $cHT=0$, we have

$$rHT=cHT=eHT=eHT=(s1, s2).$$

If $e=(0, 0, \ldots, ej, 0, \ldots)$ with $ej \neq 0$, then the definition of H gives

$$s1=ej\alpha(j-1)(k+1),\ s2=ej\alpha(j-1)(k+2).$$

Therefore, $s2/s1=\alpha j-1$. Also, $s1/\alpha(j-1)(k+1)=ej$, as claimed.

24.9 Reed-Solomon Codes

The Reed-Solomon codes, constructed in 1960, are an example of BCH codes. Because they work well for certain types of errors, they have been used in spacecraft communications and in compact discs.

Let F be a finite field with q elements and let n=q−1. A basic fact from the theory of finite fields is that F contains a primitive nth root of unity α. Choose d with 1 ≤ d<n and let

$$g(X)=(X-\alpha)(X-\alpha 2)\cdots(X-\alpha d-1).$$

This is a polynomial with coefficients in F. It generates a BCH code C over F of

length n, called a **Reed-Solomon code**.

Since $g(\alpha)=\cdots=g(\alpha d-1)=0$, the BCH bound implies that the minimum distance for C is at least d. Since g(X) is a polynomial of degree $d-1$, it has at most d nonzero coefficients. Therefore, the codeword corresponding to the coefficients of g(X) is a codeword of weight at most d. It follows that the minimum weight for C is exactly d. The dimension of C is $n-\deg(g)=n+1-d$. Therefore, a Reed-Solomon code is a cyclic $[n, n+1-d, d]$ code.

The codewords in C correspond to the polynomials

$g(X)f(X)$ with $\deg(f) \leq n-d$.

There are $qn-d+1$ such polynomials

f(X) since there are q choices for each of the n−d+1 coefficients of f(X), and thus there are qn−d+1 codewords in C. Therefore, a Reed-Solomon code is a MDS code, namely, one that makes the Singleton bound (Section 24.3) an equality.

Example

Let F=Z7={0, 1, 2, …, 6}, the integers mod 7. Then q=7 and n=q−1=6. A primitive sixth root of unity α in F is the same as a primitive root mod 7 (see Section 3.7). We may take α=3. Choose d=4. Then

$$g(X)=(X-3)(X-32)(X-33)=X3+3X2+X+6.$$

The code has generating matrix

$$G=613100061310006131.$$

There are 73=343 codewords in the code, obtained by taking all linear combinations mod 7 of the three rows of G. The minimum weight of the code is 4.

Example

Let F=GF(4)={0, 1, ω, ω2}, which was introduced in Section 3.11. Then F has 4 elements, n=q−1=3, and α=ω. Choose d=2, so

g(X)=(X−ω).

The matrix

G=ω100ω1

is a generating matrix for the code. The code contains all 16 linear combinations of the two rows of G, for example,

$\omega \cdot (\omega, 1, 0) + 1 \cdot (0, \omega, 1) = (\omega 2, 0, 1)$.

The minimum weight of the code is 2.

In many applications, errors are not randomly distributed. Instead, they occur in bursts. For example, in a CD, a scratch introduces errors in many adjacent bits. A burst of solar energy could have a similar effect on communications from a spacecraft. Reed-Solomon codes are useful in such situations.

For example, suppose we take $F=GF(28)$. The elements of F are represented as bytes of eight bits each, as in Section 3.11. We have $n=28-1=255$. Let $d=33$. The codewords are then vectors consisting of 255 bytes. There are 222 information bytes and 33

check bytes. These codewords are sent as strings of 8×255=2040 binary bits. Disturbances in the transmission will corrupt some of these bits. However, in the case of bursts, these bits will often be in a small region of the transmitted string. If, for example, the corrupted bits all lie within a string of 121 (=15×8+1) consecutive bits, there can be errors in at most 16 bytes. Therefore, these errors can be corrected (because $16<d/2$). On the other hand, if there were 121 bit errors randomly distributed through the string of 2040 bits, numerous bytes would be corrupted, and correct decoding would not be possible. Therefore, the choice of code depends on the type of errors that are expected.

24.10 The McEliece Cryptosystem

In this book, we have mostly described cryptographic systems that are based on number theoretic principles. There are many other cryptosystems that are based on other complex problems. Here we present one based on the difficulty of finding the nearest codeword for a linear binary code.

The idea is simple. Suppose you have a binary string of length 1024 that has 50 errors. There are $(.5501024) \approx 3\times1085$ possible locations for these errors, so an exhaustive search that tries all possibilities is infeasible. Suppose,

however, that you have an efficient decoding algorithm that is unknown to anyone else. Then only you can correct these errors and find the corrected string. McEliece showed how to use this to obtain a public key cryptosystem.

Bob chooses G to be the generating matrix for an (n, k) linear error correcting code C with d(C)=d. He chooses S to be a k×k matrix that is invertible mod 2 and lets P be an n×n permutation matrix, which means that P has exactly one 1 in every row and in every column, with all the other entries being 0. Define

G1=SGP.

The matrix G1 is the public key for the cryptosystem. Bob keeps S, G, P secret.

In order for Alice to send Bob a message x, she generates a random binary string e of length n that has weight t. She forms the ciphertext by computing

$y \equiv xG1+e \pmod 2$.

Bob decrypts y as follows:

1. Calculate $y1 \equiv yP-1$. (Since P is a permutation matrix, $e1=eP-1$ is still a binary string of weight t. We have $y1 \equiv xSG+e1$.)

2. Apply the error decoder for the code C to y1 to correct the “error” and obtain the codeword x1 closest to y1.

3. Compute x0 such that $x0G \equiv x1$ (in the examples we have considered, x0 is simply the first k bits of x1).

4. Compute $x \equiv x0S-1$.

The security of the system lies in the

difficulty of decoding y1 to obtain x1. There is a little security built into the system by S; however, once a decoding algorithm is known for the code generated by GP, a chosen plaintext attack allows one to solve for the matrix S (as in the Hill cipher).

To make decoding difficult, d(C) should be chosen to be large. McEliece suggested using a [1024, 512, 101] Goppa code. The **Goppa codes** have parameters of the form $n=2m$, $d=2t+1$, $k=n-mt$. For example, taking $m=10$ and $t=50$ yields the [1024, 524, 101] code just mentioned. It can correct up to 50 errors. For given values of m and t, there are in fact many inequivalent Goppa codes with these parameters. We

will not discuss these codes here except to mention that they have an efficient decoding algorithm and therefore can be used to correct errors quickly.

Example

Consider the matrix

$$G=1000110010010100100110001111,$$

which is the generator matrix for the [7, 4] Hamming code. Suppose Alice wishes to send a message

$$m=(1, 0, 1, 1)$$

to Bob. In order to do so, Bob must create an invertible matrix S and a random permutation matrix P that he will keep secret. If Bob chooses

$$S=1001110101011110$$

and

P=00100001000000000010000000100000001010000000001000.

Using these, Bob generates the public encryption matrix

G1=00110101010011111000101010100.

In order to encrypt, Alice generates her own random error vector e and calculates the ciphertext y=xG1+e. In the case of a Hamming code the error vector has weight 1. Suppose Alice chooses

e=(0, 1, 0, 0, 0, 0, 0).

Then

y=(0, 0, 0, 1, 1, 0, 0).

Bob decrypts by first calculating

y1=yP−1=(0, 0, 1, 0, 0, 0, 1).

Calculating the syndrome of y1 by applying the parity check matrix H and changing the corresponding bit yields

x1=(0, 0, 1, 0, 0, 1, 1).

Bob next forms a vector x0 such that x0G=x1, which can be done by extracting the first four components of x1, that is,

x0=(0, 0, 1, 0).

Bob decrypts by calculating

x=x0S−1=(1, 0, 1, 1),

which is the original plaintext message.

The McEliece system seems to be reasonably secure. For a discussion of its security, see [Chabaud]. A disadvantage of the system compared to

RSA, for example, is that the size of the public key G1 is rather large.

24.11 Other Topics

The field of error correcting codes is a vast subject that is explored by both the mathematical community and the engineering community. In this chapter we have only touched upon a select handful of the concepts of this field. There are many other areas of error correcting codes that we have not discussed.

Perhaps most notable of these is the study of convolutional codes. In this chapter we have entirely focused on block codes, where typically the data are segmented into blocks of a fixed length k and mapped into codewords of a fixed

length n. However, in many applications, the data are produced in a continuous fashion, and it is better to map the stream of data into a stream of coded symbols. For example, such codes have the advantage of not requiring the delay needed to observe an entire block of symbols before encoding or decoding. A good analogy is that block codes are the coding theory analogue of block ciphers, while convolutional codes are the analogue of stream ciphers.

Another topic that is very important in the study of error correcting codes is that of efficient decoding. In the case of linear codes, we presented syndrome decoding, which is more efficient than

performing a search for the nearest codeword. However, for large linear codes, syndrome decoding is still too inefficient to be practical. When BCH and Reed-Solomon codes were introduced, the decoding schemes that were originally presented were impractical for decoding more than a few errors. Later, Berlekamp and Massey provided an efficient approach to decoding BCH and Reed-Solomon codes. There is still a lot of research being done on this topic. We direct the reader to the books [Lin-Costello], [Wicker], [Gallager], and [Berlekamp] for further discussion on the subject of decoding.

We have also focused entirely on bit or

symbol errors. However, in modern computer networks, the types of errors that occur are not simply bit or symbol errors but also the complete loss of segments of data. For example, on the Internet, data are transferred over the network in chunks called packets. Due to congestion at various locations on the network, such as routers and switches, packets might be dropped and never reach their intended recipient. In this case, the recipient might notify the sender, requesting a packet to be resent. Protocols such as the Transmission Control Protocol (TCP) provide mechanisms for retransmitting lost packets.

When performing cryptography, it is

critical to use a combination of many different types of error control techniques to assure reliable delivery of encrypted messages; otherwise, the receiver might not be able to decrypt the messages that were sent.

Finally, we mention that coding theory has strong connections with various problems in mathematics such as finding dense packings of high-dimensional spheres. For more on this, see [Thompson].

24.12 Exercises

1. Two codewords were sent using the Hamming [7, 4] code and were received as 0100111 and 0101010. Each one contains at most one error. Correct the errors. Also, determine the 4-bit messages that were multiplied by the matrix G to obtain the codewords.

2. An ISBN number is incorrectly written as 0-13-116093-8. Show that this is not a correct ISBN number. Find two different valid ISBN numbers such that an error in one digit would give this number. This shows that ISBN cannot correct errors.

3. The following is a parity check matrix for a binary [n, k] code C:

   111001001001001.

   1. Find n and k.

   2. Find the generator matrix for C.

   3. List the codewords in C.

4. What is the code rate for C?

4. Let C={(0, 0, 0), (1, 1, 1)} be a binary repetition code.

    1. Find a parity check matrix for C.

    2. List the cosets and coset leaders for C.

    3. Find the syndrome for each coset.

    4. Suppose you receive the message (1, 1, 0). Use the syndrome decoding method to decode it.

5. Let C be the binary code {(0, 0, 1), (1, 1, 1), (1, 0, 0), (0, 1, 0)}.

    1. Show that C is not linear.

    2. What is d(C)? (Since C is not linear, this cannot be found by calculating the minimum weight.)

    3. Show that C satisfies the Singleton bound with equality.

6. Show that the weight function (on Fn) satisfies the triangle inequality: $wt(u+v) \leq wt(u)+wt(v)$.

7. Show that Aq(n, n)=q, where Aq(n, d) is the function defined in Section 24.3.

8. Let C be the repetition code of length n. Show that C⊥ is the parity check code of length n. (This is true for arbitrary F.)

9. Let C be a linear code and let u+C and v+C be cosets of C. Show that u+C=v+C if and only if u −v ∈ C. (Hint: To show u+C=v+C, it suffices to show that u+c ∈ v+C for every c ∈ C, and that v +c ∈ u+C for every c ∈ C. To show the opposite implication, use the fact that u ∈ u+C.)

10. Show that if C is a self-dual [n, k, d] code, then n must be even.

11. Show that g(X)=1+X+X2+⋯+Xn−1 is the generating polynomial for the [n, 1] repetition code. (This is true for arbitrary F.)

12. Let g(X)=1+X+X3 be a polynomial with coefficients in Z2.

    1. Show that g(X) is a factor of X7−1 in Z2[X].

    2. The polynomial g(X) is the generating polynomial for a cyclic code [7, 4] code C. Find the generating matrix for C.

3. Find a parity check matrix H for C.

4. Show that $G'HT=0$, where

   $G'=110100001101000011010011100 1.$

5. Show that the rows of G' generate C.

6. Show that a permutation of the columns of G' gives the generating matrix for the Hamming [7, 4] code, and therefore these two codes are equivalent.

13. Let C be the cyclic binary code of length 4 with generating polynomial $g(X)=X2+1$. Which of the following polynomials correspond to elements of C?

    $f1(X)=1+X+X3$, $f2(X)=1+X$
    $+X2+X3$, $f3(X)=X2+X3$

14. Let $g(X)$ be the generating polynomial for a cyclic code C of length n, and let $g(X)h(X)=Xn-1$. Write $h(X)=b0+b1X+\cdots+X\ell$. Show that the dual code $C\perp$ is cyclic with generating polynomial $h\tilde{}r(X)=(1/b0)(1+b\ell-1X+\cdots+b1X\ell-1+b0X\ell)$. (The factor 1/b0 is included to make the highest nonzero coefficient be 1.)

15.

1. Let C be a binary repetition code of odd length n (that is, C contains two vectors, one with all 0s and one with all 1s). Show that C is perfect. (Hint: Show that every vector lies in exactly one of the two spheres of radius $(n-1)/2$.)

2. Use (a) to show that if n is odd then $\sum_{j=0}^{(n-1)/2} \binom{n}{j} = 2^{n-1}$. (This can also be proved by applying the binomial theorem to $(1+1)^n$, and then observing that we're using half of the terms.)

16. Let $2 \le d \le n$ and let $V_q(n, d-1)$ denote the number of points in a Hamming sphere of radius $d-1$. The proof of the Gilbert-Varshamov bound constructs an (n, M, d) code with $M \ge q^n / V_q(n, d-1)$. However, this code is probably not linear. This exercise will construct a linear $[n, k, d]$ code, where k is the smallest integer satisfying $q^k \ge q^n / V_q(n, d-1)$.

    1. Show that there exists an $[n, 1, d]$ code C_1.

    2. Suppose $q^{j-1} < q^n / V_q(n, d-1)$ and that we have constructed an $[n, j-1, d]$ code C_{j-1} in F^n (where F is the finite field with q elements). Show that there is a

vector v with $d(v, c) \geq d$ for all $c \in Cj-1$.

3. Let Cj be the subspace spanned by v and Cj−1. Show that Cj has dimension j and that every element of Cj can be written in the form av+c with $a \in F$ and $c \in Cj-1$.

4. Let av+c, with $a \neq 0$, be an element of Cj, as in (c). Show that wt(av+c)=wt(v+a −1c)=d(v, −a−1c) ≥ d.

5. Show that Cj is an [n, j, d] code. Continuing by induction, we obtain the desired code Ck.

6. Here is a technical point. We have actually constructed an [n, k, e] code with e ≥ d. Show that by possibly modifying v in step (b), we may arrange that d(v, c)=d for some $c \in Cj-1$, so we obtain an [n, k, d] code.

17. Show that the Golay code G23 is perfect.

18. Let α be a root of the polynomial X3+X +1 ∈ Z2[X].

    1. Using the fact that X3+X+1 divides X7−1, show that α7=1.

2. Show that $\alpha \neq 1$.

3. Suppose that $\alpha j=1$ with $1 \leq j<7$. Then $\gcd(j, 7)=1$, so there exist integers a, b with $ja+7b=1$. Use this to show that $\alpha 1=1$, which is a contradiction. This shows that α is a primitive seventh root of unity.

19. Let C be the binary code of length 7 generated by the polynomial $g(X)=1+X2+X3+X4$. As in Section 24.8, $g(1)=g(\alpha)=0$, where α is a root of $X3+X+1$. Suppose the message (1, 0, 1, 1, 0, 1, 1) is received. It has one error. Use the procedure from Section 24.8 to correct the error.

20. Let $C \subset Fn$ be a cyclic code of length n with generating polynomial g(X). Assume $0 \neq C \neq Fn$ and $p|n$ (as in the theorem on p. 472).

    1. Show that $\deg(g) \geq 1$.

    2. Write $Xn-1=g(X)h(X)$. Let α be a primitive nth root of unity. Show that at least one of $1, \alpha, \alpha 2, \ldots, \alpha n-1$ is a root of g(X). (You may use the fact that h(X) cannot have more than deg(h) roots.)

    3. Show that $d(C) \geq 2$.

24.13 Computer Problems

1. Three codewords from the Golay code G24 are sent and you receive the vectors

   (0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1), (0, 0, 1

   Correct the errors. (The Golay matrix is stored as *golay* and the matrix B is stored in the downloadable computer files (`bit.ly/2JbcS6p`) as *golaybt*.)

2. An 11-bit message is multiplied by the generating matrix for the Hamming [15, 11] code and the resulting codeword is sent. The vector

   (0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0)

   is received. Assuming there is at most one error, correct it and determine the original 11-bit message. (The parity check matrix for the Hamming [15, 11] code is stored in the downloadable computer files (`bit.ly/2JbcS6p`) as *hammingpc*.)

Chapter 25 Quantum Techniques in Cryptography

Quantum computing is a new area of research that has only recently started to blossom. Quantum computing and quantum cryptography were born out of the study of how quantum mechanical principles might be used in performing computations. The Nobel Laureate Richard Feynman observed in 1982 that certain quantum mechanical phenomena could not be simulated efficiently on a classical computer. He suggested that the situation could perhaps be reversed by using quantum mechanics to do computations that are impossible on

classical computers. Feynman didn't present any examples of such devices, and only recently has there been progress in constructing even small versions.

In 1994 the field of quantum computing had a significant breakthrough when Peter Shor of AT&T Research Labs introduced a quantum algorithm that can factor integers in (probabilistic) polynomial time (if a suitable quantum computer is ever built). This was a dramatic breakthrough as it presented one of the first examples of a scenario in which quantum techniques might significantly outperform classical computing techniques.

In this chapter we introduce a couple of

examples from the area of quantum computing and quantum cryptography. By no means is this chapter a thorough treatment of this young field, for even as we write this chapter significant breakthroughs are being made at NIST and other places, and the field likely will continue to advance rapidly.

There are many books and expository articles being written on quantum computing. One readable account is [Rieffel-Polak].

25.1 A Quantum Experiment

Quantum mechanics is a difficult subject to explain to nonphysicists since it deals with concepts where our everyday experiences aren't applicable. In particular, the scale at which quantum mechanical phenomena take place is on the atomic level, which is something that can't be observed without special equipment. There are a few examples, however, that are accessible to us, and we now present one such example and use it to develop the mathematical formulation needed to describe some quantum computing protocols.

Since quantum mechanics is a particle-

level physics, we need particles that we are able to observe. Photons are the particles that make up light and are therefore observable (similar demonstrations using other particles, such as electrons, can be performed but require more sophisticated equipment).

In order to understand this experiment better, we recommend that you try it at home. Start with a light source and three Polaroid® filters from a camera supply store or three lenses from Polaroid sunglasses.

Label the three filters A, B, and C. Rotate them so that they have the following polarizations: horizontal, 45°, and vertically, respectively (we will explain polarization in more detail after

the experiment). Shine the light at the wall and insert filter A between the light source and the wall as in Figure 25.1. The photons coming out of the filter will have horizontal polarization. Now insert filter C as in Figure 25.2. Since filter C has vertical polarization, it filters out all of the horizontally polarized photons from filter A. Notice that no light arrives at the wall after this step, the two filters have removed all of the light components. Now for the final (and most bizarre) step, insert filter B in between filter A and C. You should observe that there is now light arriving at the wall, as depicted in Figure 25.3. This is puzzling, since filter A and C were enough to remove all of the light, yet the addition of a third filter allows

for light to reach the wall.

Figure 25.1 The Photon Experiment with Only Filter A Inserted

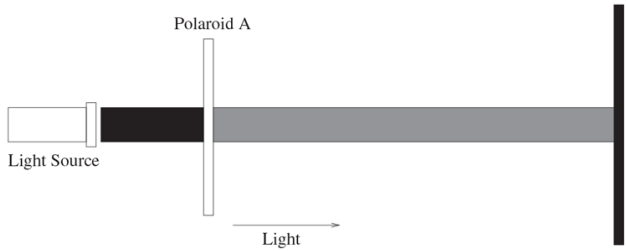


Figure 25.1 Full Alternative Text

Figure 25.2 The Photon Experiment with Filters A and C Inserted

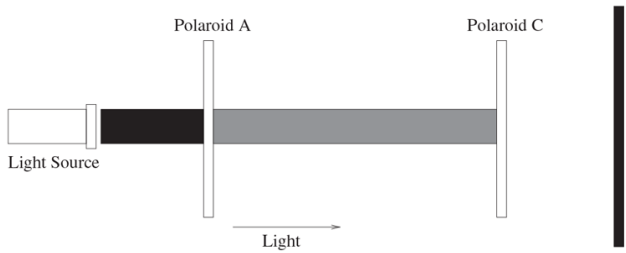


Figure 25.2 Full Alternative Text

Figure 25.3 The Photon Experiment after All Filters Have Been Inserted

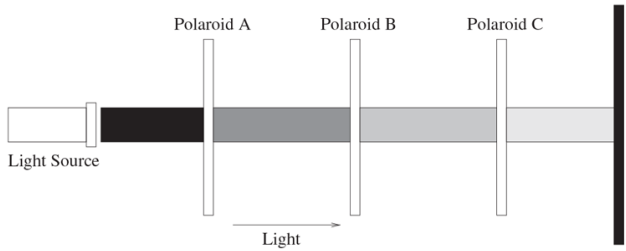


Figure 25.3 Full Alternative Text

In order to explain this demonstration, we need to discuss the concept of polarization of light.

Light is an example of an electromagnetic wave, meaning that it consists of an electric field that travels orthogonally to a corresponding magnetic field. In order to visualize this, consider the light traveling along the x-axis. Now imagine, for example, that the electric field is a wavelike function that

lies in the xz-plane. Then the corresponding magnetic field would be a wavelike function in the xy-plane. For such a scenario, the light is referred to as vertically polarized. In general, polarization refers to the direction in which the electric field lies. There is no constraint on this direction.

We will represent a photon's polarization by a unit vector in the two-dimensional complex vector space (however, for our present purposes, real numbers suffice). This vector space has a dot product given by $(a, b)\cdot(c, d)=ac^{-}+bd^{-}$, where c^{-} and d^{-} denote the complex conjugates of c and d. The square of the length of a vector (a, b) is then $(a, b)\cdot(a, b)=|a|2+|b|2$. Choose a

basis, which we shall denote $|\uparrow\rangle$ and $|\rightarrow\rangle$, for this vector space. We are choosing to use the ket (the second half of "bracket") notation from physics to represent vectors. We can think of $|\uparrow\rangle$ as being the vertical direction and $|\rightarrow\rangle$ as being horizontal. Therefore, an arbitrary polarization may be represented as $a|\uparrow\rangle+b|\rightarrow\rangle$, where a and b are complex numbers. Since we are working with unit vectors, the following property holds: $|a|2+|b|2=1$. We could just have well chosen a different orthogonal basis, for example, one corresponding to a 45° rotation: $|\nwarrow\rangle$ and $|\nearrow\rangle$.

The Polaroid filters perform a measurement of the polarity of the photon. There are two possible

outcomes: Either the photon is aligned with the filter, or it is perpendicular to the direction of the filter. If the vector a|↑⟩+b|→⟩ is measured by a vertical filter, then the probability that the photon has vertical polarity after passing through the filter is |a|2. The probability that it is measured as having horizontal polarity, and therefore not pass through, is |b|2.

Similarly, suppose we measure a vertically aligned photon with respect to a 45∘ filter. Since

|↑⟩=12|↖⟩+12|↗⟩,

the probability that the photon passes through the filter (which means that it is measured as being aligned at 45∘) is (1/2)2=1/2. Similarly, the probability that it doesn't pass through the filter

(which means that it is measured at −45°) is also 1/2.

One of the basic principles of quantum mechanics is that such a measurement forces the photon into a definite state. After being measured, the state of the photon will be changed to the result of the measurement. Therefore, if we measured the state of $a|\uparrow\rangle+b|\rightarrow\rangle$ as $|\rightarrow\rangle$, then, from that moment on, the photon will have the state $|\rightarrow\rangle$. If we then measure this photon with a $|\rightarrow\rangle$ filter, we will always observe that the photon is in the $|\rightarrow\rangle$ state; however, if we measure with a $|\uparrow\rangle$ filter, we will never observe that the photon is in the $|\uparrow\rangle$ state.

Let's now explain the interpretation of the experiment. The original light was

emitted with random polarization, so only half of the photons being emitted will pass through the $|\rightarrow\rangle$ filter, and these photons will have their state changed to $|\rightarrow\rangle$. The remaining half will be absorbed or reflected and will be changed to $|\uparrow\rangle$. When we place the vertical filter after the horizontal filter, the photons that hit it, which are in state $|\rightarrow\rangle$, will be stopped.

When we insert filter B in the middle, it corresponds to measuring with respect to $|\nearrow\rangle$, and hence those photons that had $|\rightarrow\rangle$ polarity will come out having $|\nearrow\rangle$ polarity with probability 1/2. Therefore, there has been a 4:1 reduction in the amount of photons passing through up to filter B. Now the $|\nearrow\rangle$ photons pass

through the $|\uparrow\rangle$ filter with probability 1/2 also, and so the total intensity of light arriving at the wall is 1/8th the original intensity.

25.2 Quantum Key Distribution

Now that we have set up some of the ideas behind quantum mechanics, we can use them to describe a technique for distributing bits through a quantum channel. These bits can be used to establish a key that can be used for communicating across a classical channel, or any other shared secret.

We begin by describing a quantum bit. Start with a two-dimensional complex vector space. Choose a pair of orthogonal vectors of length 1; call them $|0\rangle$ and $|1\rangle$. For example, these two vectors could be either of the two pairs

of orthogonal vectors used in the previous section. A **quantum bit**, also known as a **qubit**, is a unit vector in this vector space. For the purposes of the present discussion, we can think of a qubit as a polarized photon. We have chosen $|0\rangle$ and $|1\rangle$ as notation to conveniently represent the 0 and 1 bits, respectively. The other qubits are linear combinations of these two bits.

Since a qubit is a unit vector, it can be represented as $a|0\rangle+b|1\rangle$, where a and b are complex numbers such that $|a|2+|b|2=1$. Just as in the case for photons from the preceding section, we can measure this qubit with respect to the basis $|0\rangle$, $|1\rangle$. The probability that we observe it in the $|0\rangle$ state is $|a|2$.

Let us now examine how Alice and Bob can communicate with each other in order to establish a message. They will need two things: a quantum channel and a classical channel. A quantum channel is one through which they can exchange polarized photons that are isolated from interactions with the environment (that is, the environment doesn't alter the photons). The classical channel will be used to send ordinary messages to each other. We assume that the evil observer Eve can observe what is being sent on the classical channel and that she can observe and resend photons on the quantum channel.

Alice starts the establishment of a message by sending a sequence of bits

to Bob. They are encoded using a randomly chosen basis for each bit as follows. There are two bases: B1={|↑⟩, |→⟩} and B2={|↖⟩, |↗⟩}. If Alice chooses B1, then she encodes 0 as |↑⟩ and 1 as |→⟩, while if she chooses B2 then she encodes 0 and 1 using the two elements of B2.

Each time Alice sends a photon, Bob randomly chooses to measure with respect to either basis B1 or B2. Therefore, for each photon, he obtains an element of that choice of basis as the result of his measurement. Bob records the measurements he has made and keeps them secret. He then tells Alice the basis with which he measured each photon. Alice responds to Bob by telling

him which bases were the correct bases for the polarity of the photons that she sent. They keep the bits that used the same bases and discard the other bits. Since two bases were used, Alice and Bob will agree on roughly half of the amount of bits that Alice sent. They can then use these bits as the key for a conventional cryptographic system.

Example

Suppose Alice wants to send the bits 0, 1, 1, 1, 0, 0, 1, 0. She randomly chooses the bases
B1, B2, B1, B1, B2, B2, B1, B2.
Therefore, she sends the qubits (photons)

$|\uparrow\rangle, |\nearrow\rangle, |\rightarrow\rangle, |\rightarrow\rangle, |\nwarrow\rangle, |\nwarrow\rangle, |\rightarrow\rangle, |\nwarrow\rangle$

to Bob. He chooses the bases B2, B2, B2, B1, B2, B1, B1, B2. He measures the qubits that Alice sent and also tells Alice which bases he used. Alice tells him that the second, fourth, fifth, seventh, and eighth match her choices. These yielded measurements

$|\nearrow\rangle, |\rightarrow\rangle, |\nwarrow\rangle, |\rightarrow\rangle, |\nwarrow\rangle$

for Bob, and they correspond to the bits 1, 1, 0, 1, 0. Therefore, both Alice and Bob have the same string 1, 1, 0, 1, 0. They use 11010 as a key for future communication (for example, if they obtained a longer string, they could use the first 128 characters for an AES key).

The security behind quantum key distribution is based upon the laws of quantum mechanics and the

fundamental principle that following a measurement of a particle, that particle's state will be altered. Since an eavesdropper Eve must perform measurements in order to observe the photon transmissions between Alice and Bob, Eve will introduce errors in the data that Alice and Bob agreed upon.

Let's see how this happens. Suppose Eve measures the states of the photons transmitted by Alice and allows these measured photons to proceed onto Bob. Since these photons were measured by Eve, they will have the state that Eve observed. Eve will use the wrong basis half of the time when performing the measurement. When Bob performs his measurements, if he uses the correct

basis there will be a 25% chance that he will have measured the wrong value.

Let's examine this last statement in more detail. Suppose that Alice sends a photon corresponding to $|\rightarrow\rangle$ and that Bob uses the same basis B1 as Alice. If Eve uses B1, then the photon is passed through correctly and then Bob measures the photon correctly. However, if Eve used B2, then she will measure $|\nearrow\rangle$ and $|\nwarrow\rangle$ equally likely. The photons that pass to Bob will have one of these orientations and he will therefore half the time measure them correctly as $|\rightarrow\rangle$ and half the time incorrectly. Combining the two possible choices of basis that Eve has causes Bob to have a 25 chance of measuring the

incorrect value.

Thus, any eavesdropping introduces a higher error rate in the communication between Alice and Bob. If Alice and Bob test their data for discrepancies over the conventional channel (for example, they could send parity bits), they will detect any eavesdropping.

Actual implementations of this technique have been used to establish keys over distances of more than 100 km using conventional fiber optical cables.

25.3 Shor's Algorithm

Quantum computers are not yet a reality. The current versions can only handle a few qubits. But, if the great technical problems can be overcome and large quantum computers are built, the effect on cryptography will be enormous. In this section we give a brief glimpse at how a quantum computer could factor large integers, using an algorithm developed by Peter Shor. We avoid discussing quantum mechanics and ask the reader to believe that a quantum computer should be able to do all the operations we describe, and do them quickly. For more details, see, for example, [Ekert-Josza] or [Rieffel-

Polak].

What is a quantum computer and what does it do? First, let's look at what a classical computer does. It takes a binary input, for example, 100010, and gives a binary output, perhaps 0101. If it has several inputs, it has to work on them individually. A quantum computer takes as input a certain number of qubits and outputs some qubits. The main difference is that the input and output qubits can be linear combinations of certain basic states. The quantum computer operates on all basic states in this linear combination simultaneously. In effect, a quantum computer is a massively parallel machine.

For example, think of the basic state |

100⟩ as representing three particles, the first in orientation 1 and the last two in orientation 0 (with respect to some basis that will implicitly be fixed throughout the discussion). The quantum computer can take |100⟩ and produce some output. However, it can also take as input a normalized (that is, of length 1) linear combination of basic quantum states such as

$$\frac{1}{3}(|100\rangle+|011\rangle+|110\rangle)$$

and produce an output just as quickly as it did when working with a basic state. After all, the computer could not know whether a quantum state is one of the basic states, or a linear combination of them, without making a measurement. But such a measurement would alter the

input. It is this ability to work with a linear combination of states simultaneously that makes a quantum computer potentially very powerful.

Suppose we have a function f(x) that can be evaluated for an input x by a classical computer. The classical computer asks for an input and produces an output. A quantum computer, on the other hand, can accept as input a sum

$$\frac{1}{C}\sum x|x\rangle$$

(C is a normalization factor) of all possible input states and produce the output

$$\frac{1}{C}\sum x|x, f(x)\rangle,$$

where $|x, f(x)\rangle$ is a longer sequence of qubits, representing both x and the value

of f(x). (*Technical point:* It might be notationally better to input (1/C) ∑ | x, 00⋯⟩ in order to have some particles to change to f(x). For simplicity, we will not do this.) So we can obtain a list of all the values of f(x). This looks great, but there is a problem. If you make a measurement, you force the quantum state into the result of the measurement. You get |x0, f(x0)⟩ for some randomly chosen x0, and the other states in the output are destroyed. So, if you are going to look at the list of values of f(x), you'd better do it carefully, since you get only one chance. In particular, you probably want to apply some transformation to the output in order to put it into a more desirable form. The skill in programming a quantum

computer is in designing the computation so that the outputs you want to examine appear with much higher probability than the others. This is what is done in Shor's factorization algorithm.

25.3.1 Factoring

We want to factor n. The strategy is as follows. Recall that if we can find (nontrivial) a and r with $a^r \equiv 1 \pmod{n}$, then we have a good chance of factoring n (see the factorization method in Subsection 9.4.1). Choose a random a and consider the sequence $1, a, a^2, a^3, \ldots \pmod{n}$. If $a^r \equiv 1 \pmod{n}$, then this sequence will repeat every r terms since $a^{j+r} \equiv a^j a^r \equiv a^j \pmod{n}$. If we can measure the period of this

sequence (or a multiple of the period), we will have an r such that $ar \equiv 1 \pmod{n}$. We therefore want to design our quantum computer so that when we make a measurement on the output, we'll have a high chance of obtaining the period.

25.3.2 The Discrete Fourier Transform

We need a technique for finding the period of a periodic sequence. Classically, Fourier transforms can be used for this purpose, and they can be used in the present situation, too. Suppose we have a sequence

$$a_0, a_1, \ldots, a_{2m-1}$$

of length 2m, for some integer m.

Define the Fourier transform to be

$$F(x)=12m\sum c=02m-1e2\pi icx2mac,$$

where $0 \le x<2m$.

For example, consider the sequence

1, 3, 7, 2, 1, 3, 7, 2

of length 8 and period 4. The length divided by the period is the frequency, namely 2, which is how many times the sequence repeats. The Fourier transform takes the values

$$F(0)=26/8,\ F(2)=(-12+2i)/8,\ F(4)=6/8,\ F(6)=(-12-2i)/8,$$
$$F(1)=F(3)=F(5)=F(7)=0.$$

For example, letting $\zeta=e2\pi i/8$, we find that

$$8F(1)=1+3\zeta+7\zeta2+2\zeta3+\zeta4+3\zeta5+7\zeta6+2\zeta7.$$

Since $\zeta4=-1$, the terms cancel and we

obtain F(1)=0. The nonzero values of F occur at multiples of 2, which is the frequency.

Let's consider another example:
2, 1, 2, 1, 2, 1, 2, 1. The Fourier transform is

F(0)=12/8, F(4)=4/8,
F(1)=F(2)=F(3)=F(5)=F(6)=F(7)=0.

Here the nonzero values of F are again at the multiples of the frequency.

In general, if the period is a divisor of 2m, then all the nonzero values of F will occur at multiples of the frequency (however, a multiple of the frequency could still yield 0). See Exercise 2.

Suppose now that the period isn't a

divisor of 2m. Let's look at an example. Consider the sequence
1, 0, 0, 1, 0, 0, 1, 0. It has length 8 and almost has period 3 and frequency 3, but we stopped the sequence before it had a chance to complete the last period. In Figure 25.4, we graph the absolute value of its Fourier transform (these are real numbers, hence easier to graph than the complex values of the Fourier transform). Note that there are peaks at 0, 3, and 5. If we continued F(x) to larger values of x we would get peaks at 8, 11, 13, 16, …. The peaks are spaced at an average distance of 8/3. Dividing the length of the sequence by the average distance yields a period of 8/(8/3)=3, which agrees with our intuition.

Figure 25.4 The Absolute Value of a Discrete Fourier Transform

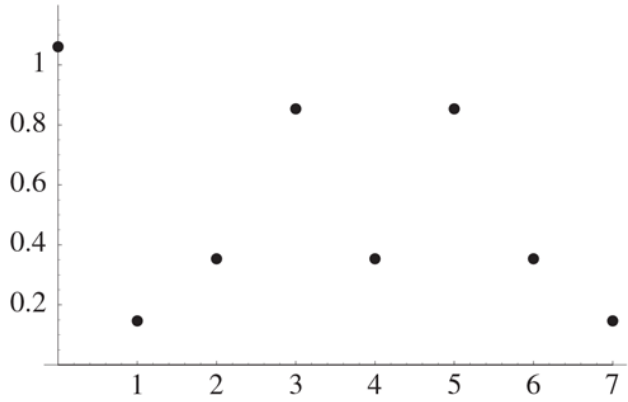


Figure 25.4 Full Alternative Text

The fact that there is a peak at 0 is not very surprising. The formula for the Fourier transform shows that the value at 0 is simply the sum of the elements in the sequence divided by the square root of the length of the sequence.

Let’s look at one more example: 1, 0, 0, 0, 0, 1, 0, 0, 0, 0 1, 0, 0, 0, 0, 1. This

sequence has 16 terms. Our intuition might say that the period is around 5 and the frequency is slightly more than 3. Figure 25.5 shows the graph of the absolute value of its Fourier transform. Again, the peaks are spaced around 3 apart, so we can say that the frequency is around 3. The period of the original sequence is therefore around 5, which agrees with our intuition.

Figure 25.5 The Absolute Value of a Discrete Fourier Transform

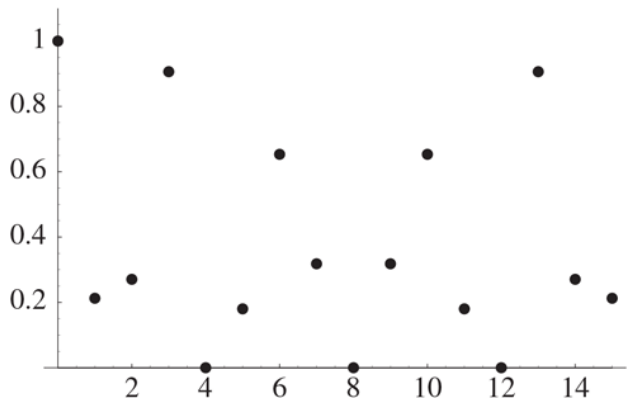


Figure 25.5 Full Alternative Text

In the first two examples, the period was a divisor of the length (namely, 8) of the sequence. We obtained nonzero values of the Fourier transform only at multiples of the frequency. In these last two examples, the period was not a divisor of the length (8 or 16) of the sequence. This introduced some “noise” into the situation. We had peaks at approximate multiples of the frequency and values close to 0 away from these peaks.

The conclusion is that the peaks of the Fourier transform occur approximately at multiples of the frequency, and the period is approximately the number of peaks. This will be useful in Shor’s

algorithm.

25.3.3 Shor's Algorithm

Choose m so that n2 ≤ 2m<2n2. We start with m qubits, all in state 0:

|000000000⟩.

As in the previous section, by changing axes, we can transform the first bit to a linear combination of |0⟩ and |1⟩, which gives us

12(|000000000⟩+|100000000⟩).

We then successively do a similar transformation to the second bit, the third bit, up through the mth bit, to obtain the quantum state

12m(|000000000⟩+|00000001⟩+|000000010⟩+⋯+|111111111⟩).

Thus all possible states of the m qubits are superimposed in this sum. For simplicity of notation, we replace each string of 0s and 1s with its decimal equivalent, so we write

$$12m(|0\rangle+|1\rangle+|2\rangle+\cdots+|2m-1\rangle).$$

Choose a random number a with $1<a<n$. We may assume $\gcd(a, n)=1$; otherwise, we have a factor of n. The quantum computer computes the function $f(x)=ax \pmod n$ for this quantum state to obtain

$$12m(|0, a0\rangle+|1, a1\rangle+|2, a2\rangle+\cdots+|2m-1, a2m-1\rangle)$$

(for ease of notation, ax is used to denote $ax \pmod n$). This gives a list of all the values of ax. However, so far we are not any better off than with a classical computer. If we measure the

state of the system, we obtain a basic state $|x0, ax0\rangle$ for some randomly chosen x0. We cannot even specify which x0 we want to use. Moreover, the system is forced into this state, obliterating all the other values of ax that have been computed. Therefore, we do not want to measure the whole system. Instead, we measure the value of the second half. Each basic piece of the system is of the form $|x, ax\rangle$, where x represents m bits and ax is represented by m/2 bits (since $ax \pmod n < n < 2^{m/2}$). If we measure these last m/2 bits, we obtain some number $u \pmod n$, and the whole system is forced into a combination of those states of the form $|x, u\rangle$ with $ax \equiv u \pmod n$:

$$1C \sum 0 \le x < 2^{max} \equiv u \pmod n |x, u\rangle,$$

where C is whatever factor is needed to make the vector have length 1 (in fact, C is the square root of the number of terms in the sum).

Example

At this point, it is probably worthwhile to have an example. Let n=21. (This example might seem simple, but it is the largest that quantum computers using Shor's algorithm can currently handle. Other algorithms are being developed that can go somewhat farther.) Since $212<29<2\cdot212$, we have m=9. Let's choose a=11, so we compute the values of $11x \pmod{21}$ to obtain

$$1512(|0,\ 1\rangle+|1,\ 11\rangle+|2,\ 16\rangle+|3,\ 8\rangle+|4,\ 4\rangle+|5,\ 2\rangle+|6,\ 1\rangle+|7,\ 11\rangle+|8,\ 16\rangle+|9,\ 8\rangle+|10,\ 4\rangle+|11,\ 2\rangle+|12,\ 1\rangle+|13,\ 11\rangle+|14,\ 16\rangle+|15,\ 8\rangle+|16,\ 4\rangle+|17,\ 2\rangle+|18,\ 1\rangle+|19,\ 11\rangle+|20,\ 16\rangle+\cdots+|508,\ 4\rangle+|509,\ 2\rangle+|510,\ 1\rangle+|511,\ 11\rangle).$$

Suppose we measure the second part and obtain 2. This means we have extracted all the terms of the form $|x,\ 2\rangle$ to obtain

$$\frac{1}{85}(|5,\ 2\rangle+|11,\ 2\rangle+|17,\ 2\rangle+|23,\ 2\rangle+\cdots+|497,\ 2\rangle+|503,\ 2\rangle+|509,\ 2\rangle).$$

For notational convenience, and since it will no longer be needed, we drop the second part to obtain

$$\frac{1}{85}(|5\rangle+|11\rangle+|17\rangle+|23\rangle+\cdots+|497\rangle+|503\rangle+|509\rangle).$$

If we now measured this system, we would simply obtain a number x such that $11x \equiv 2 \pmod{21}$. This would not be useful.

Suppose we could take two measurements. Then we would have two numbers x and y with $11x \equiv 11y \pmod{}$

21). This would yield $11x - y \equiv 1 \pmod{21}$. By the factorization method of Subsection 9.4.1, this would give us a good chance of being able to factor 21. However, we cannot take two independent measurements. The first measurement puts the system into the output state, so the second measurement would simply give the same answer as the first.

Not all is lost. Note that in our example, the numbers in our state are periodic with period 6. In general, the values of $a^x \pmod{n}$ are periodic with period r, with $a^r \equiv 1 \pmod{n}$. So suppose we are able to make a measurement that yields the period. We then have a situation where $a^r \equiv 1 \pmod{n}$, so we can hope to

factor n by the method from Subsection 9.4.1 mentioned above.

The **quantum Fourier transform** is exactly the tool we need. It measures frequencies, which can be used to find the period. If r happens to be a divisor of 2m, then the frequencies we obtain are multiples of a fundamental frequency f0, and rf0=2m. In general, r is not a divisor of 2m, so there will be some dominant frequencies, and they will be approximate multiples of a fundamental frequency f0 with rf0 ≈ 2m. This will be seen in the analysis of our example and in Figure 25.6.

Figure 25.6 The Absolute Value of g(c)

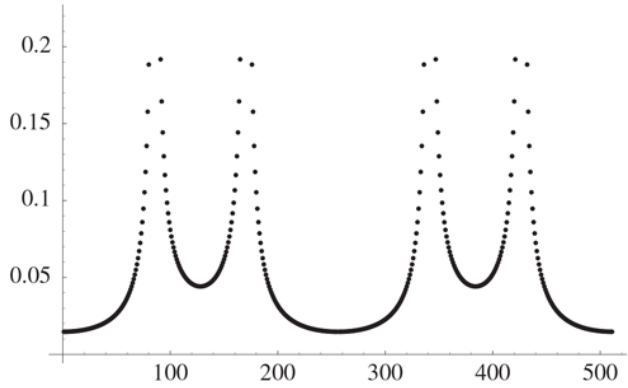


Figure 25.6 Full Alternative Text

The quantum Fourier transform is defined on a basic state $|x\rangle$ (with $0 \leq x<2m$) by

$$QFT(|x\rangle)=12m\sum c=02m-1e2\pi icx2m|c\rangle.$$

It extends to a linear combination of states by linearity:

$$QFT(a1|x1\rangle+\cdots+at|xt\rangle)=a1QFT(|x1\rangle)+\cdots+atQFT(|xt\rangle).$$

We can therefore apply QF< to our quantum state.

In our example, we compute

$$QFT<(185(|5\rangle+|11\rangle+|17\rangle+|23\rangle+\cdots+|497\rangle+|503\rangle+|509\rangle))$$

and obtain a sum

$$185\sum c=0511g(c)|c\rangle$$

for some numbers g(c).

The number g(c) is given by

$$g(c)=1512\sum 0 \le x<512x \equiv 5 \pmod 6 e2\pi icx512,$$

which is the discrete Fourier transform of the sequence

0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, …, 0, 0, 0, 0, 0, 1, 0, 0.

Therefore, the peaks of the graph of the absolute value of g should correspond to the frequency of the sequence, which should be around $512/6 \approx 85$. The graph in Figure 25.6 is a plot of |g|.

There are sharp peaks at c=0, 85, 171,

256, 341, 427 (the ones at 0 and 256 do not show up on the graph since they are centered at one value; see below). These are the dominant frequencies mentioned previously. The values of g near the peak at c=341 are

338339340341342343344345 0.305 0.439 0.773 3.111 1.567 0.631 0.398 0.291

The behavior near c=85, 171, and 427 is similar. At c=0 and 256, we have g(0)=3.756, while all the nearby values of c have g(c) ≈ 0.015.

The peaks are approximately at multiples of the fundamental frequency f0=85. Of course, we don't really know this yet, since we haven't made any measurements.

Now we measure the quantum state of

this Fourier transform. Recall that if we start with a linear combination of states $a1|x1\rangle+\cdots+a<|x<\rangle$ normalized such that $\sum|aj|2=1$, then the probability of of obtaining $|xk\rangle$ is $|ak|2$. More generally, if we don't assume $\sum|aj|2=1$, the probability is

$$|ak|2/\sum|aj|2.$$

In our example,

$$3.1112/\sum|aj|2 \approx .114,$$

so if we sample the Fourier transform, the probability is around $4\times.114=.456$ that we obtain one of c=85, 171, 341, 427. Let's suppose this is the case; say we get c=427. We know, or at least expect, that 427 is approximately a multiple of the frequency f0 that we're looking for:

$$427 \approx jf0$$

for some j. Since rf0 ≈ 2m=512, we divide to obtain

$$427512 \approx jr.$$

Note that 427/512 ≈ .834 ≈ 5/6. Since we must have r ≤ ϕ(21)<21, a reasonable guess is that r=6 (see the following discussion of continued fractions).

In general, Shor showed that there is a high chance of obtaining a value of c/2m with

$$|c2m−jr|<12m+1<12n2,$$

for some j. The method of continued fractions will find the unique (see Exercise 3) value of j/r with r<n satisfying this inequality.

In our example, we take r=6 and check that ar=116 ≡ 1 (mod 21).

We want to use the factorization method of Subsection 9.4.1 to factor 21. Recall that this method writes r=2km with m odd, and then computes b0 ≡ am (mod n). We then successively square b0 to get b1, b2, …, until we reach 1 (mod n). If bu is the last bi ≡ 1 (mod n), we compute gcd(bu−1, n) to get a factor (possibly trivial) of n.

In our example, we write 6=2·3 (a power of 2 times an odd number) and compute (in the notation of Subsection 9.4.1)

$$b0 \equiv 113 \equiv 8 \pmod{21} \quad b1 \equiv 116 \equiv 1 \pmod{21}$$
$$\gcd(b0-1,\ 21)=\gcd(7,\ 21)=7,$$

so we obtain $21=7\cdot 3$.

In general, once we have a candidate for r, we check that $a^r \equiv 1 \pmod{n}$. If not, we were unlucky, so we start over with a new a and form a new sequence of quantum states. If $a^r \equiv 1 \pmod{n}$, then we use the factorization method from Subsection 9.4.1. If this fails to factor n, start over with a new a. It is very likely that, in a few attempts, a factorization of n will be found.

We now say more about continued fractions. In Chapter 3, we outlined the method of continued fractions for finding rational numbers with small denominator that approximate real numbers. Let's apply the procedure to the real number 427/512. We have

427512=0+11+15+142+12.

This yields the approximating rational numbers

0, 1, 56, 211253, 427512.

Since we know the period in our example is less than n=21, the best guess is the last denominator less than n, namely r=6.

In general, we compute the continued fraction expansion of c/2m, where c is the result of the measurement. Then we compute the approximations, as before. The last denominator less than n is the candidate for r.

25.3.4 Final Words

The capabilities of quantum computers

and quantum algorithms are of significant importance to economic and government institutions. Many secrets are protected by cryptographic protocols. Quantum cryptography's potential for breaking these secrets as well as its potential for protecting future secrets has caused this new research field to grow rapidly over the past few years.

Although the first full-scale quantum computer is probably many years off, and there are still many who are skeptical of its possibility, quantum cryptography has already succeeded in transmitting secure messages over a distances of more than 100 km, and quantum computers have been built that

can handle a (very) small number of qubits. Quantum computation and cryptography have already changed the manner in which computer scientists and engineers perceive the capabilities and limits of the computer. Quantum computing has rapidly become a popular interdisciplinary research area and promises to offer many exciting new results in the future.

25.4 Exercises

1. Consider the sequence 20, 21, 22, … (mod 15).

    1. What is the period of this sequence?

    2. Suppose you want to use Shor's algorithm to factor n=15. What value of m would you take?

    3. Suppose the measurement in Shor's algorithm yields c=192. What value do you obtain for r? Does this agree with part (a)?

    4. Use the value of r from part (c) to factor 15.

2.

    1. Let $0<s \leq m$. Fix an integer c0 with $0 \leq c0<2s$. Show that

    $$\sum 0 \leq c<2mc \equiv c0 \pmod{2s} e2\pi icx2m=0$$

    if $x \equiv 0 \pmod{2m-s}$ and $=2m-se2\pi ixc0/2m$ if $x \equiv 0 \pmod{2m-s}$. (Hint: Write $c=c0+j2s$ with $0 \leq j<2m-s$,

factor e2πixc0/2m off the sum, and recognize what's left as a geometric sum.)

2. Suppose a0, a1, …, a2m−1 is a sequence of length 2m such that ak=ak+j2s for all j, k. Show that the Fourier transform F(x) of this sequence is 0 whenever $x \equiv 0 \pmod{2m-s}$.

   This shows that if the period of a sequence is a divisor of 2m then all the nonzero values of F occur at multiples of the frequency (namely, 2m−s).

3.

1. Suppose j/r and j1/r1 are two distinct rational numbers, with 0<r<n and 0<r1<n. Show that

   |j1r1−jr|g>1n2.

2. Suppose, as in Shor's algorithm, that we have

   |c2m−jr|<12n2 and |c2m−j1r1|<12n2.

   Show that j/r=j1/r1.

Appendix A Mathematica® Examples

These computer examples are written in Mathematica. If you have Mathematica available, you should try some of them on your computer. If Mathematica is not available, it is still possible to read the examples. They provide examples for several of the concepts of this book. For information on getting started with Mathematica, see Section A.1. To download a Mathematica notebook that contains these commands, go to

```
bit.ly/2u5R7dW
```

A.1 Getting Started with Mathematica

1. Download the Mathematica notebook crypto.nb that you find using the links starting at `bit.ly/2u5R7dW`

2. Open Mathematica, and then open crypto.nb using the menu options under File on the command bar at the top of the Mathematica window. (Perhaps this is done automatically when you download it; it depends on your computer settings.)

3. With crypto.nb in the foreground, click (left button) on Evaluation on the command bar. A menu will appear. Move the arrow down to the line Evaluate Notebook and click (left button). This evaluates the notebook and loads the necessary functions. Ignore any warning messages about spelling. They occur because a few functions have similar names.

4. Go to the command bar at the top and click on File. Move the arrow down to New and left click. Then left click on Notebook. A new notebook will appear on top of crypto.nb. However, all the

commands of crypto.nb will still be working.

5. If you want to give the new notebook a name, use the File command and scroll down to Save As.... Then save under some name with a .nb at the end.

6. You are now ready to use Mathematica. If you want to try something easy, type 1+2*3+4a^5 and then press the Shift and Enter keys simultaneously. Or, if your keyboard has a number pad with Enter, probably on the right side of the keyboard, you can press that (without the Shift). The result 1031 should appear (it's 1+2·3+45).

7. Turn to the Computer Examples Section A.3. Try typing in some of the commands there. The outputs should be the same as that in the examples. Remember to press Shift Enter (or the numeric Enter) to make Mathematica evaluate an expression.

8. If you want to delete part of your notebook, simply move the arrow to the line at the right edge of the window and click the left button. The highlighted part can be deleted by clicking on Edit on the top command bar, then clicking on Cut on the menu that appears.

9. Save your notebook by clicking on File on the command bar, then clicking on Save on the menu

that appears.

10. Print your notebook by clicking on File on the command bar, then clicking on Print on the menu that appears. (You will see the advantage of opening a new notebook in Step 4; if you didn't open one, then all the commands in crypto.nb will also be printed.)

11. If you make a mistake in typing in a command and get an error message, you can edit the command and hit Shift Enter to try again. You don't need to retype everything.

12. Look at the commands available through the command bar at the top. For example, Format then Style allows you to change the type font on any cell that has been highlighted (by clicking on its bar on the right side).

13. If you are looking for help or a command to do something, try the Help command. Note that the commands that are built into Mathematica always start with capital letters. The commands that are coming from crypto.nb start with small letters and will not be found via Help.

A.2 Some Commands

The following are some Mathematica commands that are used in the Computer Examples. The commands that start with capital letters, such as `EulerPhi`, are built into Mathematica. The ones that start with small letters, such as `addell`, have been written specially for this text and are in the Mathematica notebook available at

```
bit.ly/2u5R7dW
```

`addell[{x,y}, {u,v}, b, c, n]` finds the sum of the points {x, y} and {u, v} on the elliptic curve y2 ≡ x3+bx+c (mod n), where n is odd.

`affinecrypt[txt,m,n]` affine encryption of txt using mx+n.

`allshifts[txt]` gives all 26 shifts of txt.

`ChineseRemainder[{a,b,...},{m,n,...}]` gives a solution to the simultaneous congruences
$x \equiv a \pmod{m}$, $x \equiv b \pmod{n}$, ….

`choose[txt,m,n]` lists the characters in txt in positions congruent to n (mod m).

`coinc[txt,n]` the number of matches between txt and txt displaced by n.

`corr[v]` the dot product of the vector v with the 26 shifts of the alphabet frequency vector.

`EulerPhi[n]` computes $\phi(n)$ (don't try very large values of n).

`ExtendedGCD[m,n]` computes the gcd of m and n along with a solution of mx +ny=gcd.

`FactorInteger[n]` factors n.

`frequency[txt]` lists the number of occurrences of each letter a through z in txt.

`GCD[m,n]` is the gcd of m and n.

`Inverse[M]` finds the inverse of the matrix M.

`lfsr[c,k,n]` gives the sequence of n bits produced by the recurrence that has

coefficients given by the vector c. The initial values of the bits are given by the vector k.

`lfsrlength[v,n]` tests the vector v of bits to see if it is generated by a recurrence of length at most n.

`lfsrsolve[v,n]` given a guess n for the length of the recurrence that generates the binary vector v, it computes the coefficients of the recurrence.

`Max[v]` is the largest element of the vector v.

`Mod[a,n]` is the value of a (mod n).

`multell[{x,y}, m, b, c, n]` computes m times the point {x, y} on the elliptic

curve y2 ≡ x3+bx+c (mod n).

`multsell[{x,y}, m, b, c, n]` lists the first m multiples of the point {x, y} on the elliptic curve y2 ≡ x3+bx+c (mod n).

`NextPrime[x]` gives the next prime>x.

`num2text0[n]` changes a number n to letters. The successive pairs of digits must each be at most 25; a is 00, z is 25.

`num2text[n]` changes a number n to letters. The successive pairs of digits must each be at most 26; *space* is 00, a is 01, z is 26.

`PowerMod[a,b,n]` computes ab (mod n).

`PrimitiveRoot[p]` finds a primitive root

for the prime p.

`shift[txt,n]` shifts txt by n.

`txt2num0[txt]` changes txt to numbers, with a=00, …, z=25.

`txt2num[txt]` changes txt to numbers, with *space* = 00, a=01, …, z=26.

`vigenere[txt,v]` gives the Vigenère encryption of txt using the vector v.

`vigvec[txt,m,n]` gives the frequencies of the letters a through z in positions congruent to n (mod m).

A.3 Examples for Chapter 2

Example 1

A shift cipher was used to obtain the ciphertext `kddmu`. Decrypt it by trying all possibilities.

In[1]:= `allshifts["kddkmu"]`

```
kddkmu
leelnv
mffmow
nggnpx
ohhoqy
piiprz
qjjqsa
rkkrtb
sllsuc
tmmtvd
unnuwe
```

```
voovxf
wppwyg
xqqxzh
yrryai
zsszbj
attack
buubdl
cvvcem
dwwdfn
exxego
fyyfhp
gzzgiq
haahjr
ibbiks
jccjlt
```

As you can see, *attack* is the only word that occurs on this list, so that was the plaintext.

Example 2

Encrypt the plaintext message *cleopatra* using the affine function 7x+8:

In[2]:=`affinecrypt["cleopatra", 7, 8]`

Out[2]=`whkcjilxi`

Example 3

The ciphertext *mzdvezc* was encrypted using the affine function 5x+12. Decrypt it.

SOLUTION

First, solve $y \equiv 5x+12 \pmod{26}$ for x to obtain $x \equiv 5-1(y-12)$. We need to find the inverse of 5 (mod 26):

In[3]:= `PowerMod[5, -1, 26]`

Out[3]= `21`

Therefore, $x \equiv 21(y-12) \equiv 21y-12 \cdot 21$.

To change −12·21 to standard form:

In[4]:= `Mod[-12*21, 26]`

Out[4]= `8`

Therefore, the decryption function is $x \equiv 21y+8$. To decrypt the message:

In[5]:= `affinecrypt["mzdvezc", 21, 8]`

Out[5]= `anthony`

In case you were wondering, the plaintext was encrypted as follows:

In[6]:= `affinecrypt["anthony", 5, 12]`

Out[6]= `mzdvezc`

Example 4

Here is the example of a Vigenère cipher from the text. Let's see how to produce the data that was used in Section 2.3 to decrypt it. For convenience, we've already stored the ciphertext under the name *vvhq*.

In[7]:= `vvhq`

Out[7]=

```
vvhqwvvrhmusgjgthkihtssejchlsfcbgvwcrlryqtfsvg
hbltspisprdxljsveeghlqwkasskuwepwqtwvspgoelkcq
wgoviokhkazkqkxzgyhcecmeiujoqkwfwvefqhkijrclrl
twlauqrhwdmwlgusgikkflryvcwvspgpmlkassjvoqxegg
ikvrdrygfrjljslveggveyggeiapuuisfpbtgnwwmuczrv
```

Find the frequencies of the letters in the ciphertext:

In[8]:= `frequency[vvhq]`

Out[8]=

```
{{a, 8}, {b, 5}, {c, 12}, {d, 4},
{e, 15}, {f, 10}, {g, 27}, {h, 16},
{i, 13}, {j, 14}, {k, 17}, {l, 25},
{m, 7}, {n, 7}, {o, 5}, {p, 9}, {q,
14}, {r, 17}, {s, 24}, {t, 8}, {u,
12}, {v, 22}, {w, 22}, {x, 5}, {y,
8}, {z, 5}}
```

Let’s compute the coincidences for displacements of 1, 2, 3, 4, 5, 6:

In[9]:= `coinc[vvhq, 1]`

Out[9]= `14`

In[10]:= `coinc[vvhq, 2]`

Out[10]= `14`

In[11]:= `coinc[vvhq, 3]`

Out[11]= `16`

In[12]:= `coinc[vvhq, 4]`

Out[12]= `14`

In[13]:= `coinc[vvhq, 5]`

Out[13]= `24`

In[14]:= `coinc[vvhq, 6]`

Out[14]= `12`

We conclude that the key length is probably 5. Let's look at the 1st, 6th, 11th, ... letters (namely, the letters in positions congruent to 1 mod 5):

In[15]:= `choose[vvhq, 5, 1]`

Out[15]=

```
vvuttcccqgcunjtpjgkuqpknjkygkkgcjfqrkqjrqudukvpk
```

In[16]:= `frequency[%]`

Out[16]= `{{a, 0}, {b, 0}, {c, 7}, {d, 1}, {e, 1}, {f, 2}, {g, 9}, {h, 0}, {i, 1}, {j, 8}, {k, 8}, {l, 0}, {m, 0}, {n, 3}, {o, 0}, {p, 4}, {q, 5}, {r, 2}, {s, 0}, {t, 3}, {u, 6}, {v, 5}, {w, 1}, {x, 0}, {y, 1}, {z, 0}}`

To express this as a vector of frequencies:

In[17]:= `vigvec[vvhq, 5, 1]`

Out[17]= `{0, 0, 0.104478, 0.0149254, 0.0149254, 0.0298507, 0.134328, 0, 0.0149254, 0.119403, 0.119403, 0, 0, 0.0447761, 0, 0.0597015, 0.0746269, 0.0298507, 0, 0.0447761, 0.0895522,`

```
0.0746269, 0.0149254, 0, 0.0149254, 0}
```

The dot products of this vector with the displacements of the alphabet frequency vector are computed as follows:

In[18]:= `corr[%]`

Out[18]=

```
{0.0250149, 0.0391045, 0.0713284,
0.0388209, 0.0274925, 0.0380149,
0.051209, 0.0301493, 0.0324776,
0.0430299, 0.0337761, 0.0298507,
0.0342687, 0.0445672, 0.0355522,
0.0402239, 0.0434328, 0.0501791,
0.0391791, 0.0295821, 0.0326269,
0.0391791, 0.0365522, 0.0316119,
0.0488358, 0.0349403}
```

The third entry is the maximum, but sometimes the largest entry is hard to locate. One way to find it is

In[19]:= `Max[%]`

Out[19]= `0.0713284`

Now it is easy to look through the list and find this number (it usually occurs only once). Since it occurs in the third position, the first shift for this Vigenère cipher is by 2, corresponding to the letter *c*. A procedure similar to the one just used (using *vigvec[vvhq, 5,2],..., vigvec[vvhq,5,5]*) shows that the other shifts are probably 14, 3, 4, 18. Let's check that we have the correct key by decrypting.

In[20]:= `vigenere[vvhq, -{2, 14, 3, 4, 18}]`

Out[20]=

```
themethodusedforthepreparationandreadingofcode
extremeandatthesametimeimpossibleoftranslation
eeasewithwhichthekeymaybechangedisanotherpoint
fthiscodebythosedesiringtotransmitimportantmes
estdangeroftheirmessagesbeingreadbypoliticalor
```

For the record, the plaintext was originally encrypted by the command

In[21]:= `vigenere[%, {2, 14, 3, 4, 18}]`

Out[21]=

```
vvhqwvvrhmusgjgthkihtssejchlsfcbgvwcrlryqtfsvg
hbltspisprdxljsveeghlqwkasskuwepwqtwvspgoelkcq
wgoviokhkazkqkxzgyhcecmeiujoqkwfwvefqhkijrclrl
twlauqrhwdmwlgusgikkflryvcwvspgpmlkassjvoqxegg
ikvrdrygfrjljslveggveyggeiapuuisfpbtgnwwmuczrv
```

A.4 Examples for Chapter 3

Example 5

Find gcd(23456, 987654).

In[1]:= `GCD[23456, 987654]`

Out[1]= `2`

Example 6

Solve 23456x+987654y=2 in integers x, y.

In[2]:= `ExtendedGCD[23456, 987654]`

Out[2]= `{2, {-3158, 75}}`

This means that 2 is the gcd and $23456 \cdot (-3158) + 987654 \cdot 75 = 2$.

Example 7

Compute $234 \cdot 456 \pmod{789}$.

In[3]:=
```
Mod[234*456, 789]
```

Out[3]=
```
189
```

Example 8

Compute
$234567876543 \pmod{565656565}$.

In[4]:=
```
PowerMod[234567, 876543,
565656565]
```

Out[4]=
```
473011223
```

Example 9

Find the multiplicative inverse of 87878787 (mod 9191919191).

In[5]:= `PowerMod[87878787, -1, 9191919191]`

Out[5]= `7079995354`

Example 10

Solve $7654x \equiv 2389 \pmod{65537}$.

SOLUTION

Here is one way. It corresponds to the method in Section 3.3. We calculate 7654^{-1} and then multiply it by 2389:

In[6]:= `PowerMod[7654, -1, 65537]`

Out[6]= `54637`

In[7]:= `Mod[%*2389, 65537]`

Out[7]= `43626`

Example 11

Find x with

$x \equiv 2 \pmod{78}$, $x \equiv 5 \pmod{97}$, $x \equiv 1 \pmod{119}$.

SOLUTION

In[8]:= `ChineseRemainder[{2, 5, 1}, {78, 97, 119}]`

Out[8]= `647480`

We can check the answer:

In[9]:= `Mod[647480, {78, 97, 119}]`

Out[9]= `{2, 5, 1}`

Example 12

Factor 123450 into primes.

In[10]:= `FactorInteger[123450]`

Out[10]= `{{2, 1}, {3, 1}, {5, 2}, {823, 1}}`

This means that 123450=2131528231.

Example 13

Evaluate $\phi(12345)$.

In[11]:= `EulerPhi[12345]`

Out[11]= `6576`

Example 14

Find a primitive root for the prime 65537.

In[12]:= `PrimitiveRoot[65537]`

Out[12]= `3`

Therefore, 3 is a primitive root for 65537.

Example 15

Find the inverse of the matrix 1312354153627168^{10} (mod 999).

SOLUTION

First, invert the matrix without the mod:

In[13]:= `Inverse[{{13, 12, 35}, {41, 53, 62}, {71, 68, 10}}]`

Out[13]=
`{{368634139, -226034139, 111134139}, {-399234139`

We need to clear the 34139 out of the denominator, so we evaluate 1/34139 mod 999:

In[14]:= `PowerMod[34139, -1, 999]`

Out[14]= `410`

Since 410·34139 ≡ 1 (mod 999), we multiply the inverse matrix by 410·34139 and reduce mod 999 in order to remove the denominators without changing anything mod 999:

In[15]:= `Mod[410*34139*%%, 999]`

Out[15]= `{{772, 472, 965}, {641, 516, 851}, {150, 133, 149}}`

Therefore, the inverse matrix mod 999 is 772472965641516851150133149.

In many cases, it is possible to determine by inspection the common denominator that must be removed. When this is not the case, note that the determinant of the original matrix will always work as a common denominator.

Example 16

Find a square root of 26951623672 mod the prime p=98573007539.

SOLUTION

Since p ≡ 3 (mod 4), we can use the

Proposition of Section 3.9:

In[16]:=
```
PowerMod[26951623672,
(98573007539 + 1)/4, 98573007539]
```

Out[16]=
```
98338017685
```

The other square root is minus this one:

In[17]:=
```
Mod[-%, 98573007539]
```

Out[17]=
```
234989854
```

Example 17

Let n=34222273=9803·3491. Find all four solutions of
x2 ≡ 19101358 (mod 34222273).

SOLUTION

First, find a square root mod each of the two prime factors, both of which are congruent to 3 (mod 4):

In[18]:= `PowerMod[19101358, (9803 + 1)/4, 9803]`

Out[18]= `3998`

In[19]:= `PowerMod[19101358, (3491 + 1)/4, 3491]`

Out[19]= `1318`

Therefore, the square roots are congruent to ±3998 (mod 9803) and are congruent to ±1318 (mod 3491). There are four ways to combine these using the Chinese remainder theorem:

In[20]:= `ChineseRemainder[ {3998, 1318`

```
}, {9803, 3491 }]
```

Out[20]= `43210`

In[21]:= `ChineseRemainder[ {-3998, 1318 }, {9803, 3491 }]`

Out[21]= `8397173`

In[22]:= `ChineseRemainder[ {3998, -1318 }, {9803, 3491 }]`

Out[22]= `25825100`

In[23]:= `ChineseRemainder[ {-3998, -1318}, {9803, 3491}]`

Out[23]= `34179063`

These are the four desired square roots.

A.5 Examples for Chapter 5

Example 18

Compute the first 50 terms of the recurrence

$$x_{n+5} \equiv x_n + x_{n+2} \pmod 2.$$

The initial values are 0, 1, 0, 0, 0.

SOLUTION

The vector of coefficients is {1, 0, 1, 0, 0} and the initial values are given by the vector {0, 1, 0, 0, 0}. Type

In[1]:=
```
lfsr[{1, 0, 1, 0, 0}, {0, 1, 0, 0, 0}, 50]
```

Out[1]= `{0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1}`

Example 19

Suppose the first 20 terms of an LFSR sequence are 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1. Find a recurrence that generates this sequence.

SOLUTION

First, we find the length of the recurrence. The command *lfsrlength[v, n]* calculates the determinants mod 2 of the first n matrices that appear in the procedure in Section 5.2:

In[2]:=

```
lfsrlength[{1, 0, 1, 0, 1, 1, 1, 0,
0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0,
1}, 10]
     {1, 1}
     {2, 1}
     {3, 0}
     {4, 1}
     {5, 0}
     {6, 1}
     {7, 0}
     {8, 0}
     {9, 0}
     {10, 0}
```

The last nonzero determinant is the sixth one, so we guess that the recurrence has length 6. To find the coefficients:

In[3]:= `lfsrsolve[{1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1}, 6]`

Out[3]= `{1, 0, 1, 1, 1, 0}`

This gives the recurrence as

$$x_{n+6} \equiv x_n + x_{n+2} + x_{n+3} + x_{n+4} \pmod 2.$$

Example 20

The ciphertext 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0 was produced by adding the output of a LFSR onto the plaintext mod 2 (i.e., XOR the plaintext with the LFSR output). Suppose you know that the plaintext starts 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0. Find the rest of the plaintext.

SOLUTION

XOR the ciphertext with the known part of the plaintext to obtain the beginning of the LFSR output:

In[4]:=
```
Mod[{1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0} + {0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1}, 2]
```

```
Out[4]= {1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1}
```

This is the beginning of the LFSR output. Now let's find the length of the recurrence:

In[5]:=
```
lfsrlength[%, 8]
```

```
{1, 1}
{2, 0}
{3, 1}
{4, 0}
{5, 1}
```

```
{6, 0}
{7, 0}
{8, 0}
```

We guess the length is 5. To find the coefficients of the recurrence:

In[6]:= `lfsrsolve[%%, 5]`

Out[6]= `{1, 1, 0, 0, 1}`

Now we can generate the full output of the LFSR using the coefficients we just found plus the first five terms of the LFSR output:

In[7]:= `lfsr[{1, 1, 0, 0, 1}, {1, 0, 0, 1, 0}, 40]`

Out[7]=`{1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0}`

When we XOR the LFSR output with the ciphertext, we get back the plaintext:

In[8]:=
```
Mod[% + {0, 1, 1, 0, 1, 0, 1, 0,
1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1,
0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0,
0, 1, 0, 1, 1, 0}, 2]
```

Out[8]=
```
{1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0,
0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
0, 0, 0, 0}
```

This is the plaintext.

A.6 Examples for Chapter 6

Example 21

The ciphertext

22, 09, 00, 12, 03, 01, 10, 03, 04, 08, 01, 17

was encrypted using a Hill cipher with matrix

1234567810.

Decrypt it.

SOLUTION

A matrix abcd is entered as {{a, b}, {c, d}}. Type M.N to multiply matrices M and N. Type v.M to multiply

a vector v on the right by a matrix M.

First, we need to invert the matrix mod 26:

In[1]:= `Inverse[{{ 1,2,3},{ 4,5,6}, {7,8,10}}]`

Out[1]= `{{-23, -43, 1}, {23,113, -2}, {1, -2, 1}}`

Since we are working mod 26, we can't stop with numbers like 2/3. We need to get rid of the denominators and reduce mod 26. To do so, we multiply by 3 to extract the numerators of the fractions, then multiply by the inverse of 3 mod 26 to put the "denominators" back in (see Section 3.3):

In[2]:= `%*3`

Out[2]= `{{-2, -4, 3}, {-2, 11, -6}, {3, -6, 3}}`

In[3]:= `Mod[PowerMod[3, -1, 26]*%, 26]`

Out[3]= `{{8,16,1}, {8,21,24}, {1,24,1}}`

This is the inverse of the matrix mod 26. We can check this as follows:

In[4]:= `Mod[%.{{1, 2, 3}, {4, 5, 6}, {7, 8, 10}}, 26]`

Out[4]= `{{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}`

To decrypt, we break the ciphertext into blocks of three numbers and multiply each block on the right by the inverse matrix we just calculated:

In[5]:= `Mod[{22, 09, 00}.%%, 26]`

Out[5]= `{14, 21, 4}`

In[6]:= `Mod[{12, 03, 01}.%%%, 26]`

Out[6]= `{17, 19, 7}`

In[7]:= `Mod[{10, 03, 04}.%%%%, 26]`

Out[7]= `{4, 7, 8}`

In[8]:= `Mod[{08, 01, 17}.%%%%%, 26]`

Out[8]= `{11, 11, 23}`

Therefore, the plaintext is 14, 21, 4, 17, 19, 7, 4, 7, 8, 11, 11, 23. This can be changed back to letters:

In[9]:=

```
num2txt0[142104171907040708111123]
```

Out[9]= `overthehillx`

Note that the final `x` was appended to the plaintext in order to complete a block of three letters.

A.7 Examples for Chapter 9

Example 22

Suppose you need to find a large random prime of 50 digits. Here is one way. The function *NextPrime[x]* finds the next prime greater than x. The function *Random[Integer,{a,b}]* gives a random integer between a and b. Combining these, we can find a prime:

In[1]:= `NextPrime[Random[Integer, {10a^49, 10a^50 }]]`

Out[1]=
`73050570031667109175215303340488313456708913284 2`

If we repeat this procedure, we should

get another prime:

In[2]:= `NextPrime[Random[Integer, {10a^49, 10a^50 }]]`

Out[2]=
`9747640769493130325572432604058614414534105456831`

Example 23

Suppose you want to change the text *hellohowareyou* to numbers:

In[3]:= `txt2num1["hellohowareyou"]`

Out[3]= `8051212150815230118052515 21`

Note that we are now using a=1, b=2, …, z=26, since otherwise *a*'s at the beginnings of messages would disappear. (A more efficient procedure

would be to work in base 27, so the numerical form of the message would be 8+5·27+12·272+… +21·2713=874952215023845549 51. Note that this uses fewer digits.)

Now suppose you want to change it back to letters:

In[4]:=

```
num2txt1[805121215081523011805251521]
```

Out[4]= `hellohowareyou`

Example 24

Encrypt the message *hi* using RSA with n=823091 and e=17.

SOLUTION

First, change the message to numbers:

In[5]:= `txt2num1["hi"]`

Out[5]= `809`

Now, raise it to the eth power mod n:

In[6]:= `PowerMod[%, 17, 823091]`

Out[6]= `596912`

Example 25

Decrypt the ciphertext in the previous problem.

SOLUTION

First, we need to find the decryption exponent d. To do this, we need to find

$\phi(823091)$. One way is as follows:

```
In[7]:= EulerPhi[823091]

Out[7]= 821184
```

Another way is to factor n as $p \cdot q$ and then compute $(p-1)(q-1)$:

```
In[8]:= FactorInteger[823091]

Out[8]= { {659, 1 }, {1249, 1 } }

In[9]:= 658*1248

Out[9]= 821184
```

Since $de \equiv 1 \pmod{\phi(n)}$, we compute the following (note that we are finding the inverse of e mod $\phi(n)$, not mod n):

In[10]:= `PowerMod[17, -1, 821184]`

Out[10]= `48305`

Therefore, d=48305. To decrypt, raise the ciphertext to the dth power mod n:

In[11]:= `PowerMod[596912, 48305, 823091]`

Out[11]= `809`

Finally, change back to letters:

In[12]:= `num2txt1[809]`

Out[12]= `hi`

Example 26

Encrypt *hellohowareyou* using RSA with n=823091 and e=17.

SOLUTION

First, change the plaintext to numbers:

In[13]:= `txt2num1["hellohowareyou"]`

Out[13]= `8051212150815230118052515 21`

Suppose we simply raised this to the eth power mod n:

In[14]:= `PowerMod[%, 17, 823091]`

Out[14]= `447613`

If we decrypt (we know d from Example 25), we obtain

In[15]:= `PowerMod[%, 48305, 823091]`

Out[15]= `628883`

This is not the original plaintext. The reason is that the plaintext is larger than n, so we have obtained the plaintext mod n:

In[16]:=

```
Mod[805121215081523011805251521, 823091]
```

Out[16]= `628883`

We need to break the plaintext into blocks, each less than n. In our case, we use three letters at a time:

805121215081523011805251521

In[17]:= `PowerMod[80512, 17, 823091]`

Out[17]= `757396`

In[18]:= `PowerMod[121508, 17, 823091]`

Out[18]= `164513`

In[19]:= `PowerMod[152301, 17, 823091]`

Out[19]= `121217`

In[20]:= `PowerMod[180525, 17, 823091]`

Out[20]= `594220`

In[21]:= `PowerMod[1521, 17, 823091]`

Out[21]= `442163`

The ciphertext is therefore 757396164513121217594220442163. Note that there is no reason to change this back to letters. In fact, it doesn't correspond to any text with letters.

Decrypt each block individually:

In[22]:= `PowerMod[757396, 48305, 823091]`

Out[22]= `80512`

In[23]:= `PowerMod[164513, 48305, 823091]`

Out[23]= `121508`

Etc.

We'll now do some examples with large numbers, namely the numbers in the RSA Challenge discussed in Section 9.5. These are stored under the names *rsan, rsae, rsap, rsaq*:

In[24]:= `rsan`

Out[24]=

```
1143816257578888676692357799761466120102182967
7069352457338978305971235639587050589890751475
```

In[25]:= `rsae`

Out[25]= `9007`

Example 27

Encrypt each of the messages *b, ba, bar, bard* using *rsan* and *rsae*.

In[26]:= `PowerMod[num1["b"], rsae, rsan]`

Out[26]=

```
7094675846761266859837016499155078618287633106
8678226171649720012215533234846201405328798758
```

In[27]:= `PowerMod[txt2num1["ba"], rsae, rsan]`

Out[27]=

```
350451306089751003250117094498719542737882047527621759806027962270538031565564773352033671780
```

In[28]:= `PowerMod[txt2num1["bar"], rsae, rsan]`

Out[28]=

```
448145128638551010760045308594921093424295316000843645986880405953102818312822586362580298780
```

In[29]:= `PowerMod[txt2num1["bard"], rsae, rsan]`

Out[29]=

```
242380777851116664232028625120903173934852129556054323297179804928958073445752663026449873980
```

Observe that the ciphertexts are all the same length. There seems to be no easy way to determine the length of the corresponding plaintext.

Example 28

Using the factorization rsan=rsap·rsaq, find the decryption exponent for the RSA Challenge, and decrypt the ciphertext (see Section 9.5).

SOLUTION

First we find the decryption exponent:

In[30]:=
```
rsad=PowerMod[rsae,-1,
(rsap-1)*(rsaq-1)];
```

Note that we use the final semicolon to avoid printing out the value. If you want

to see the value of *rsad*, see Section 9.5, or don't use the semicolon. To decrypt the ciphertext, which is stored as *rsaci*, and change to letters:

In[31]:=
```
num2txt1[PowerMod[rsaci, rsad, rsan]]
```

Out[31]=
```
the magic words are squeamish ossifrage
```

Example 29

Encrypt the message *rsaencryptsmessageswell* using *rsan* and *rsae*.

In[32]:=
```
PowerMod[txt2num1["rsaencryptsmessageswell"], rsae, rsan]
```

Out[32]=

```
946394203490022593163058235392494964146409699
719506542543655849060139663288177535392831126
```

Example 30

Decrypt the preceding ciphertext.

SOLUTION

Fortunately, we know the decryption exponent *rsad*. Therefore, we compute

In[33]:= `PowerMod[%, rsad, rsan]`

Out[33]=
`1819010514031825162019130519190107051923051212`

In[34]:= `num2txt1[%]`

Out[34]= `rsaencryptsmessageswell`

Suppose we lose the final 4 of the ciphertext in transmission. Let's try to decrypt what's left (subtracting 4 and dividing by 10 is a mathematical way to remove the 4):

In[35]:= `PowerMod[(%%% - 4)/10, rsad, rsan]`

Out[35]=

```
47952999173195988664902352629525486409113633898841230037348796965779425411715895692126791262
```

If we try to change this to letters, we get a long error message. A small error in the plaintext completely changes the decrypted message and usually produces garbage.

Example 31

Suppose we are told that n=113137712755903125 67 is the product of two primes and that ϕ(n)=1131377111876087444 00. Factor n.

SOLUTION

We know (see Section 9.1) that p and q are the roots of X2−(n−ϕ(n)+1)X+n. Therefore, we compute

In[36]:=
```
Roots[Xa^2 -
(11313771275590312567 -
1131377118760874440 0 + 1)*X +
11313771275590312567 == 0, X]
```

Out[36]=
```
X == 128781017 | |
 X == 87852787151
```

Therefore, n=128781017·87852787151. We also could have used the quadratic formula to find the roots.

Example 32

Suppose we know *rsae* and *rsad*. Use these to factor *rsan*.

SOLUTION

We use the ar ≡ 1 factorization method from Section 9.4. First write rsae·rsad −1=2sm with m odd. One way to do this is first to compute rsae·rsad−1, then keep dividing by 2 until we get an odd number:

In[37]:= `rsae*rsad - 1`

Out[37]=

```
961034419617782266156919023359583834109854129
155985575087352659156174898557342995131594680
```

In[38]:= `%/2`

Out[38]=

```
480517209808891133078459511679791917054927064
577992787543676329578087449278671497565797340
```

In[39]:= `%/2`

Out[39]=

```
240258604904445566539229755839895958527463532
288996393771838164789043724639335748782898670
```

We continue this way for six more steps

until we get

Out[45]=

```
375404070163196197717546493499837435199161769157655686526849713248288081974896210747327917205
```

This number is m. Now choose a random integer a. Hoping to be lucky, we choose 13. As in the ar ≡ 1 factorization method, we compute

In[46]:= `PowerMod[13, %, rsan]`

Out[46]=

```
275743685070065305922434948688471611984230957030183109839862370800529338092984795490192643587
```

Since this is not ±1 (mod rsan), we

successively square it until we get ±1:

In[47]:= `PowerMod[%, 2, rsan]`

Out[47]=

```
48318960321928515580138476418723034554104099060
54996412582955636035266156108686431194298574070
```

In[48]:= `PowerMod[%, 2, rsan]`

Out[48]=

```
78172814154877356579141928058754000021948787050
15181839742056013275521913487560944732073516480
```

In[49]:= `PowerMod[%, 2, rsan]`

Out[49]=

```
428361912025087287421992990405829002029762229109444518239462186379470569442055101392992293086
```

In[50]:= `PowerMod[%, 2, rsan]`

Out[50]= `1`

Since the last number before the 1 was not ±1 (mod rsan), we have an example of $x \equiv \pm 1 \pmod{\text{rsan}}$ with $x2 \equiv 1$. Therefore, gcd(x−1, rsan) is a nontrivial factor of *rsan*:

In[51]:= `GCD[%% - 1, rsan]`

Out[51]=

```
32769132993266709549961988190834461413177642967
```

This is *rsaq*. The other factor is obtained

by computing *rsan/rsaq*:

In[52]:= `rsan/%`

Out[52]=

```
34905295108476509491478496199038981334177646384
```

This is *rsap*.

Example 33

Suppose you know that

$$1508834755694512 \equiv 168875705328582 \pmod{205611444308117}.$$

Factor 205611444308117.

SOLUTION

We use the Basic Principle of Section

9.4.

In[53]:=

```
GCD[150883475569451-16887570532858,2056114443081
```

Out[53]= `23495881`

This gives one factor. The other is

In[54]:= `205611444308117/%`

Out[54]= `8750957`

We can check that these factors are actually primes, so we can't factor any further:

In[55]:= `PrimeQ[%%]`

Out[55]= `True`

In[56]:= `PrimeQ[%%]`

Out[56]= `True`

Example 34

Factor
n=37687557542639485559998999289787323 9
by the p−1 method.

SOLUTION

Let’s choose our bound as B=100, and
let’s take a=2, so we compute
2100! (mod n):

In[57]:=
PowerMod[2,Factorial[100],37687557542639485559998999289787323 9]

Out[57]=

```
3696766783019563319394221062511995l2
```

Then we compute the gcd of 2100!−1 and n:

In[58]:=

```
GCD[% - 1,
37687557542639485559998999289787323 9]
```

Out[58]=

```
430553161739796481
```

This is a factor p. The other factor q is

In[59]:=

```
37687557542639485559998999289787323 9/%
```

Out[59]=

```
875328783798732119
```

Let’s see why this worked. The factorizations of p−1 and q−1 are

In[60]:=

```
FactorInteger[4305531617397964 81 - 1]
```

Out[60]= `{{2, 18 }, {3, 7 }, {5, 1 }, {7, 4 }, {11, 3 }, {47, 1 }}`

In[61]:=

```
FactorInteger[875328783798732119 - 1]
```

Out[61]= `{{2, 1 }, {61, 1 }, {20357, 1 }, {39301, 1 }, {8967967, 1 }}`

We see that 100! is a multiple of p−1, so $2^{100!} \equiv 1 \pmod{p}$. However, 100! is not a multiple of q−1, so it is likely that $2^{100!} \equiv 1 \pmod{q}$. Therefore, both $2^{100!}-1$ and pq have p as a factor, but only pq has q as a factor. It follows that the gcd is p.

A.8 Examples for Chapter 10

Example 35

Let's solve the discrete log problem $2x \equiv 71 \pmod{131}$ by the Baby Step-Giant Step method of Subsection 10.2.2. We take N=12 since N2>p−1=130 and we form two lists. The first is $2j \pmod{1}31$ for $0 \leq j \leq 11$:

In[1]:=
```
Do[Print[n, " " , PowerMod[2, n, 131]], {n, 0, 11}]
```

Out[1]=
```
0 1
```

```
1 2
2 4
```

```
3 8
4 16
5 32
6 64
7 128
8 125
9 119
10 107
11 83
```

The second is $71 \cdot 2^{-j} \pmod{131}$ for $0 \le j \le 11$:

In[2]:=
```
Do[Print[n, " " ,
Mod[71*PowerMod[2, -12*n, 131], 131]],
{n, 0, 11}]
```

Out[2]=
```
> 0 71
```

```
1 17
2 124
3 26
4 128
5 86
6 111
7 93
```

```
8 85
9 96
10 130
11 116
```

The number 128 is on both lists, so we see that 27 ≡ 71·2−12*4 (mod 131). Therefore,

71 ≡ 27+4*12 ≡ 255 (mod 131).

A.9 Examples for Chapter 12

Example 36

Suppose there are 23 people in a room. What is the probability that at least two have the same birthday?

SOLUTION

The probability that no two have the same birthday is $\prod_{i=1}^{22}(1-i/365)$ (note that the product stops at i=22, not i=23). Subtracting from 1 gives the probability that at least two have the same birthday:

In[1]:=
```
1 - Product[1. - i/365, {i, 22}]
```

Out[1]= `0.507297`

Note that we used 1. in the product instead of 1 without the decimal point. If we had omitted the decimal point, the product would have been evaluated as a rational number (try it, you'll see).

Example 37

Suppose a lazy phone company employee assigns telephone numbers by choosing random seven-digit numbers. In a town with 10,000 phones, what is the probability that two people receive the same number?

In[2]:= `1 - Product[1. - i/10a^7, {i, 9999}]`

Out[2]= `0.99327`

Note that the number of phones is about three times the square root of the number of possibilities. This means that we expect the probability to be high, which it is. From Section 12.1, we have the estimate that if there are around 2(ln2)107 ≈ 3723 phones, there should be a 50% chance of a match. Let's see how accurate this is:

In[3]:= `1 - Product[1. - i/10a^7, i, 3722]`

Out[3]= `0.499895`

A.10 Examples for Chapter 17

Example 38

Suppose we have a (5, 8) Shamir secret sharing scheme. Everything is mod the prime p=987541. Five of the shares are

(9853, 853), (4421, 4387), (6543, 1234), (93293, 78428), (12398, 7563).

Find the secret.

SOLUTION

One way: First, find the Lagrange interpolating polynomial through the five points:

In[1]:= `InterpolatingPolynomial[ {`
`{9853, 853 }, {4421, 4387 }, {6543, 1234`
`}, {93293, 78428 }, {12398, 7563 } }, x]`

Out[1]=

```
853+(-17672716+(+24069879538347560+(-84649159205
-4959003720134640533754 7(-93293+x)13378864151099
(-6543+x))(-4421+x)) (-9853+x)
```

Now evaluate at x=0 to find the constant term (use /.x–>0 to evaluate at x=0):

In[2]:= %/. `x - > 0`

Out[2]=

```
204484326154044983230114592433944282591222981069
```

We need to change this to an integer mod 987541, so we find the multiplicative inverse of the denominator:

In[3]:= `PowerMod[Denominator[%], -1, 987541]`

Out[3]= `509495`

Now, multiply times the numerator to get the desired integer:

In[4]:= `Mod[Numerator[%%]*%, 987541]`

Out[4]= `678987`

Therefore, 678987 is the secret.

A.11 Examples for Chapter 18

Example 39

Here is a game you can play. It is essentially the simplified version of poker over the telephone from Section 18.2. There are five cards: ten, jack, queen, king, ace. They are shuffled and disguised by raising their numbers to a random exponent mod the prime 24691313099. You are supposed to guess which one is the ace. To start, pick a random exponent. We use the semicolon after *khide* so that we cannot cheat and see what value of k is being

used.

In[1]:= `k = khide;`

Now, shuffle the disguised cards (their numbers are raised to the
kth power mod p and then randomly permuted):

In[2]:= `shuffle`

Out[2]= `{14001090567, 16098641856, 23340023892, 20919427041, 7768690848}`

These are the five cards (yours will differ from these because the k and the random shuffle will be different). None looks like the ace; that's because their numbers have been raised to powers mod the prime. Make a guess anyway.

Let’s see if you’re correct.

In[3]:= `reveal[%]`

Out[3]= `{ten, ace, queen, jack, king}`

Let’s play again:

In[4]:= `k = khide;`

In[5]:= `shuffle`

Out[5]= `{13015921305, 14788966861, 23855418969, 22566749952, 8361552666}`

Make your guess (note that the numbers are different because a different random exponent was used). Were you lucky?

In[6]:= `reveal[%]`

Out[6]= `{ten, queen, ace, king, jack}`

Perhaps you need some help. Let's play one more time:

In[7]:= `k = khide;`

In[8]:= `shuffle`

Out[8]= `{13471751030, 20108480083, 8636729758, 14735216549, 11884022059}`

We now ask for advice:

In[9]:= `advise[%]`

Out[9]= `3`

We are advised that the third card is the ace. Let's see (note that %% is used to refer to the next to last output):

In[10]:=
```
reveal[%%]
```

Out[10]=
```
{jack, ten, ace, queen, king}
```

How does this work? Read the part on "How to Cheat" in Section 18.2. Note that if we raise the numbers for the cards to the (p−1)/2 power mod p, we get

In[11]:=
```
PowerMod[{200514, 10010311,
1721050514, 11091407, 10305},
(24691313099 - 1)/ 2, 24691313099]
```

Out[11]=
```
{1, 1, 1, 1, 24691313098}
```

Therefore, only the ace is a quadratic nonresidue mod p.

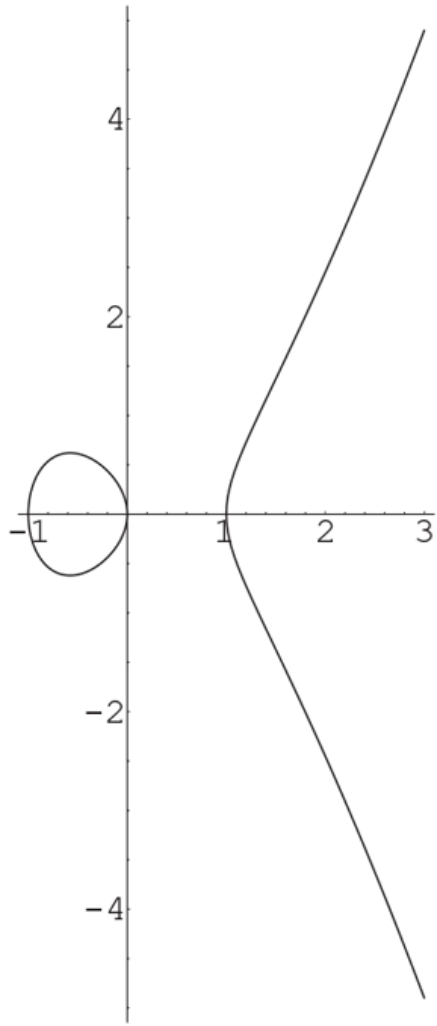
A.12 Examples for Chapter 21

Example 40

All of the elliptic curves we work with in this chapter are elliptic curves mod n. However, it is helpful to use the graphs of elliptic curves with real numbers in order to visualize what is happening with the addition law, for example, even though such pictures do not exist mod n. Therefore, let's graph the elliptic curve $y2=x(x-1)(x+1)$. We'll specify that $-1 \le x \le 3$ and $-y \le y \le 5$:

In[1]:=
```
ContourPlot[ya^2 == x*(x - 1)*(x + 1), {x, -1, 3 }, {y, -5, 5 }]
```

Graphics



Full Alternative Text

Example 41

Add the points (1, 3) and (3, 5) on the

elliptic curve $y2 \equiv x3+24x+13 \pmod{29}$.

In[2]:= `addell[ {1, 3 }, {3, 5 }, 24, 13, 29]`

Out[2]= `{26, 1 }`

You can check that the point (26, 1) is on the curve:
$263+24 \cdot 26+13 \equiv 12 \pmod{29}$.

Example 42

Add (1, 3) to the point at infinity on the curve of the previous example.

In[3]:= `addell[ {1, 3 }, {"infinity", "infinity" }, 24, 13, 29]`

Out[3]= `{1, 3 }`

As expected, adding the point at infinity to a point P returns the point P.

Example 43

Let P=(1, 3) be a point on the elliptic curve y2 ≡ x3+24x+13 (mod 29). Find 7P.

In[4]:= `multell[ {1, 3 }, 7, 24, 13, 29]`

Out[4]= `{15, 6 }`

Example 44

Find k(1, 3) for k=1, 2, 3, …, 40 on the curve of the previous example.

In[5]:= `multsell[ {1, 3 }, 40, 24, 13, 29]`

Out[5]= {1,{1,3},2,{11,10},3,{23,28},4,
{0,10},5,{19,7},6,{18,19},7, {15,6},8,
{20,24},9,{4,12},10,{4,17},11,{20,5},12,
{15,23},13,{18,10}, 14,{19,22},15,
{0,19}, 16,{23,1},17,{11,19},18,
{1,26},19, {infinity,infinity},20,
{1,3},21,{11,10}, 22,{23,28},23,{0,10},
24,{19,7}, 25,{18,19},26,{15,6},27,
{20,24},28,{4,12},29,{4,17}, 30,
{20,5},31,{15,23},32,{18,10},33,{19,22},
34,{0,19},35,{23,1},36, {11,19},37,
{1,26}, 38,{infinity,infinity},39,
{1,3},40,{11,10}}

Notice how the points repeat after every 19 multiples.

Example 45

The previous four examples worked mod the prime 29. If we work mod a composite number, the situation at infinity becomes more complicated since we could be at infinity mod both

factors or we could be at infinity mod one of the factors but not mod the other. Therefore, we stop the calculation if this last situation happens and we exhibit a factor. For example, let's try to compute 12P, where P=(1, 3) is on the elliptic curve y2 ≡ x3−5x+13 (mod 209):

In[6]:= `multell[ {1, 3 }, 12, -5, 13, 11*19]`

Out[6]= `{factor=, 19 }`

Now let's compute the successive multiples to see what happened along the way:

In[7]:= `multsell[ {1, 3 }, 12, -5, 13, 11*19]`

Out[7]= `1,{{1,3},2,{91,27},3,`

```
{118,133},4,{148,182},5,{20,35}, 6,
{factor=, 19}}
```

When we computed 6P, we ended up at infinity mod 19. Let's see what is happening mod the two prime factors of 209, namely 19 and 11:

In[8]:= `multsell[{1,3}, 12, -5, 13, 19]`

Out[8]= `1,{{1,3},2,{15,8},3,{4,0},4,`
`{15,11},5,{1,16}, 6,{infinity,infinity},`
`7,{1,3},8,{15,8},9,{4,0},10,{15,11}, 11,`
`{1,16}, 12,{infinity,infinity}}`

In[9]:= `multsell[ {1, 3 }, 20, -5, 13,`
`11]`

Out[9]= `1,{{1,3},2,{3,5},3,{8,1},4,`
`{5,6},5,{9,2},6,{6,10},7,{2,0}, 8,`
`{6,1},9,{9,9},10,{5,5},11,{8,10},12,`
`{3,6},13,{1,8}, 14,`
`{infinity,infinity},15,{1,3}, 16,`
`{3,5},17,{8,1},18,{5,6}, 19,{9,2},20,`

```
{6,10}}
```

After six steps, we were at infinity mod 19, but it takes 14 steps to reach infinity mod 11. To find 6P, we needed to invert a number that was 0 mod 19 and nonzero mod 11. This couldn't be done, but it yielded the factor 19. This is the basis of the elliptic curve factorization method.

Example 46

Factor 193279 using elliptic curves.

SOLUTION

First, we need to choose some random elliptic curves and a point on each curve. For example, let's take P=(2, 4)

and the elliptic curve

$$y2 \equiv x3-10x+b \pmod{193279}.$$

For P to lie on the curve, we take b=28. We'll also take

$$y2 \equiv x3+11x-11,\ P=(1,\ 1)y2 \equiv x3+17x-14,\ P=(1,\ 2).$$

Now we compute multiples of the point P. We do the analog of the p−1 method, so we choose a bound B, say B=12, and compute B!P.

In[10]:= `multell[{2,4}, Factorial[12], -10, 28, 193279]`

Out[10]= `{factor=, 347}`

In[11]:= `multell[{1,1}, Factorial[12], 11, -11, 193279]`

Out[11]= `{13862, 35249}`

In[12]:= `multell[{1, 2}, Factorial[12], 17, -14, 193279]`

Out[12]= `{factor=, 557}`

Let's analyze in more detail what happened in these examples.

On the first curve, 266P ends up at infinity mod 557 and 35P is infinity mod 347. Since 266=2·7·19, it has a prime factor larger than B=12, so B!P is not infinity mod 557. But 35 divides B!, so B!P is infinity mod 347.

On the second curve, 356P=infinity mod 347 and 561P=infinity mod 557. Since 356=4·89 and 561=3·11·17, we don't expect to find the factorization with this curve.

The third curve is a surprise. We have 331P=infinity mod 347 and 272P=infinity mod 557. Since 331 is prime and 272=16·17, we don't expect to find the factorization with this curve. However, by chance, an intermediate step in the calculation of B!P yielded the factorization. Here's what happened. At one step, the program required adding the points (184993, 13462) and (20678, 150484). These two points are congruent mod 557 but not mod 347. Therefore, the slope of the line through these two points is defined mod 347 but is 0/0 mod 557. When we tried to find the multiplicative inverse of the denominator mod 193279, the gcd algorithm yielded the factor 557. This phenomenon is fairly rare.

Example 47

Here is how to produce the example of an elliptic curve ElGamal cryptosystem from Section 21.5. For more details, see the text. The elliptic curve is $y2 \equiv x3+3x+45 \pmod{8831}$ and the point is G=(4, 11). Alice's message is the point Pm=(5, 1743).

Bob has chosen his secret random number aB=3 and has computed aBG:

In[13]:= `multell[{4, 11}, 3, 3, 45, 8831]`

Out[13]= `{413, 1808}`

Bob publishes this point. Alice chooses the random number k=8 and computes kG and Pm+k(aBG):

In[14]:= `multell[{4, 11}, 8, 3, 45, 8831]`

Out[14]= `{5415, 6321}`

In[15]:= `addell[{5, 1743}, multell[{413, 1808}, 8, 3, 45, 8831], 3, 45, 8831]`

Out[15]= `{6626, 3576}`

Alice sends (5415, 6321) and (6626, 3576) to Bob, who multiplies the first of these points by aB:

In[16]:= `multell[{5415, 6321}, 3, 3, 45, 8831]`

Out[16]= `{673, 146}`

Bob then subtracts the result from the last point Alice sends him. Note that he

subtracts by adding the point with the second coordinate negated:

In[17]:= `addell[{6626, 3576}, {673, -146}, 3, 45, 8831]`

Out[17]= `{5, 1743}`

Bob has therefore received Alice's message.

Example 48

Let's reproduce the numbers in the example of a Diffie-Hellman key exchange from Section 21.5: The elliptic curve is y2 ≡ x3+x+7206 (mod 7211) and the point is G=(3, 5). Alice chooses her secret NA=12 and Bob chooses his secret NB=23. Alice calculates

In[18]:= `multell[{3, 5}, 12, 1, 7206, 7211]`

Out[18]= `{1794, 6375}`

She sends (1794,6375) to Bob. Meanwhile, Bob calculates

In[19]:= `multell[{3, 5}, 23, 1, 7206, 7211]`

Out[19]= `{3861, 1242}`

and sends (3861,1242) to Alice. Alice multiplies what she receives by NA and Bob multiplies what he receives by NB:

In[20]:= `multell[{3861, 1242}, 12, 1, 7206, 7211]`

Out[20]= `{1472, 2098}`

In[21]:= `multell[{1794, 6375}, 23, 1, 7206, 7211]`

Out[21]= `{1472, 2098}`

Therefore, Alice and Bob have produced the same key.

Appendix B Maple® Examples

These computer examples are written in Maple. If you have Maple available, you should try some of them on your computer. If Maple is not available, it is still possible to read the examples. They provide examples for several of the concepts of this book. For information on getting started with Maple, see Section B.1. To download a Maple notebook that contains the necessary commands, go to

```
bit.ly/2TzKFec
```

B.1 Getting Started with Maple

1. Download the Maple notebook math.mws that you find using the links starting at `bit.ly/2TzKFec`

2. Open Maple (on a Linux machine, use the command `xmaple`; on most other systems, click on the Maple icon)), then open math.mws using the menu options under File on the command bar at the top of the Maple window. (Perhaps this is done automatically when you download it; it depends on your computer settings.)

3. With math.mws in the foreground, press the Enter or Return key on your keyboard. This will load the functions and packages needed for the following examples.

4. You are now ready to use Maple. If you want to try something easy, type 1+2*3+4^5; and then press the Return/Enter key. The result 1031 should appear (it's 1+2·3+45).

5. Go to the Computer Examples in Section B.3. Try

typing in some of the commands there. The outputs should be the same as those in the examples. Press the Return or Enter key to make Maple evaluate an expression.

6. If you make a mistake in typing in a command and get an error message, you can edit the command and hit Return or Enter to try again. You don't need to retype everything.

7. If you are looking for help or a command to do something, try the Help menu on the command bar at the top. If you can guess the name of a function, there is another way. For example, to obtain information on gcd, type `?gcd` and Return or Enter.

B.2 Some Commands

The following are some Maple commands that are used in the examples. Some, such as `phi`, are built into Maple. Others, such as `addell`, are in the Maple notebook available at

```
bit.ly/2TzKFec
```

If you want to suppress the output, use a colon instead.

The argument of a function is enclosed in round parentheses. Vectors are enclosed in square brackets. Entering `matrix(m,n,[a,b,c,...,z])` gives the m×n matrix with first row `a,b,` ... and last

row `...z`. To multiply two matrices A and B, type `evalm(A&*B)`.

If you want to refer to the previous output, use %. The next-to-last output is %%, etc. Note that % refers to the most recent output, not to the last displayed line. If you will be referring to an output frequently, it might be better to name it. For example, `g:=phi(12345)` defines `g` to be the value of $\phi(12345)$. Note that when you are assigning a value to a variable in this way, you should use a colon before the equality sign. Leaving out the colon is a common cause of hard-to-find errors.

Exponentiation is written as `a^b`. However, we will need to use modular exponentiation with very large

exponents. In that case, use `a&^bmodn`. For modular exponentiation, you might need to use a \ between & and `a^`. Use the right arrow to escape from the exponent.

Some of the following commands require certain Maple packages to be loaded via the commands

```
with(numtheory), with(linalg),
with(plots), with(combinat)
```

These are loaded when the math.mws notebook is loaded. However, if you want to use a command such as `nextprime` without loading the notebook, first type `with(numtheory):` to load the package (once for the whole session). Then you can use functions such as

`nextprime`, `isprime`, etc. If you type `with(numtheory)` without the colon, you'll get a list of the functions in the package, too.

The following are some of the commands used in the examples. We list them here for easy reference. To see how to use them, look at the examples. We have used `txt` to refer to a string of letters. Such strings should be enclosed in quotes (`"string"`).

`addell([x,y], [u,v], b, c, n)` finds the sum of the points (x, y) and (u, v) on the elliptic curve $y2 \equiv x3+bx+c \pmod{n}$. The integer n should be odd.

`affinecrypt(txt,m,n)` is the affine encryption of `txt` using mx+n.

`allshifts(txt)` gives all 26 shifts of `txt`.

`chrem([a,b,...], [m,n,...])` gives a solution to the simultaneous congruences $x \equiv a \pmod{m}$, $x \equiv b \pmod{n}$, ...

`choose(txt,m,n)` lists the characters in `txt` in positions that are congruent to n (mod m).

`coinc(txt,n)` is the number of matches between `txt` and `txt` displaced by n.

`corr(v)` is the dot product of the vector v with the 26 shifts of the alphabet frequency vector.

`phi(n)` computes $\phi(n)$ (don't try very large values of n).

`igcdex(m,n,'x','y')` computes the gcd of m and n along with a solution of mx +ny=gcd. To get `x` and `y`, type `x;y` on this or a subsequent command line.

`ifactor(n)` factors n.

`frequency(txt)` lists the number of occurrences of each letter `a` through `z` in `txt`.

`gcd(m,n)` is the gcd of m and n.

`inverse(M)` finds the inverse of the matrix M.

`lfsr(c,k,n)` gives the sequence of n bits produced by the recurrence that has coefficients given by the vector c. The initial values of the bits are given by the

vector k.

`lfsrlength(v,n)` tests the vector v of bits to see if it is generated by a recurrence of length at most n.

`lfsrsolve(v,n)` computes the coefficients of a recurrence, given a guess n for the length of the recurrence that generates the binary vector v.

`max(v)` is the largest element of the list v.

`a mod n` is the value of a (mod n).

`multell([x,y], m, b, c, n)` computes m times the point (x, y) on the elliptic curve $y2 \equiv x3+bx+c \pmod{n}$.

`multsell([x,y], m, b, c, n)` lists the first

m multiples of the point (x, y) on the elliptic curve $y2 \equiv x3+bx+c \pmod n$.

`nextprime(x)` gives the next prime >x.

`num2text(n)` changes a number n to letters. The successive pairs of digits must each be at most 26 *space* is 00, a is 01, z is 26.

`primroot(p)` finds a primitive root for the prime p.

`shift(txt,n)` shifts `txt` by n.

`text2num(txt)` changes `txt` to numbers, with `space=00, a=01, ..., z=25`.

`vigenere(txt,v)` gives the Vigenère encryption of `txt` using the vector v as

the key.

`vigvec(txt,m,n)` gives the frequencies of the letters `a` through `z` in positions congruent to n (mod m).

B.3 Examples for Chapter 2

Example 1

A shift cipher was used to obtain the ciphertext `kddkmu`. Decrypt it by trying all possibilities.

```
> allshifts("kddkmu")
                    "kddkmu"
                    "leelnv"
                    "mffmow"
                    "nggnpx"
                    "ohhoqy"
                    "piiprz"
                    "qjjqsa"
                    "rkkrtb"
                    "sllsuc"
                    "tmmtvd"
                    "unnuwe"
                    "voovxf"
```

```
"wppwyg"
"xqqxzh"
"yrryai"
"zsszbj"
"attack"
"buubdl"
"cvvcem"
"dwwdfn"
"exxego"
"fyyfhp"
"gzzgiq"
"haahjr"
"ibbiks"
"jccjlt"
```

As you can see, `attack` is the only word that occurs on this list, so that was the plaintext.

Example 2

Encrypt the plaintext message `cleopatra` using the affine function 7x+8:

```
> affinecrypt("cleopatra", 7, 8)

                        "whkcjilxi"
```

Example 3

The ciphertext `mzdvezc` was encrypted using the affine function 5x+12. Decrypt it.

SOLUTION

First, solve $y \equiv 5x+12 \pmod{26}$ for x to obtain $x \equiv 5^{-1}(y-12)$. We need to find the inverse of 5 (mod 26):

```
>5&^   (-1)mod26

                    21
```

(On some computers, the a^ doesn't

work. Instead, type a backslash \ and then a^. Use the right arrow key to escape from the exponent before typing `mod`. For some reason, a space is needed before a parenthesis in an exponent.)

Therefore, $x \equiv 21(y-12) \equiv 21y-12 \cdot 21$. To change $-12 \cdot 21$ to standard form:

```
>-12*21 mod26

                    8
```

Therefore, the decryption function is $x \equiv 21y+8$. To decrypt the message:

```
> affinecrypt("mzdvezc", 21, 8)

                       "anthony"
```

In case you were wondering, the plaintext was encrypted as follows:

```
> affinecrypt("anthony", 5, 12)

                        "mzdvezc"
```

Example 4

Here is the example of a Vigenère cipher from the text. Let's see how to produce the data that was used in Section 2.3 to decrypt it. For convenience, we've already stored the ciphertext under the name `vvhq`.

```
> vvhq

vvhqwvvrhmusgjgthkihtssejchlsfcbgvwcrlryqtfsvg
hbltspisprdxljsveeghlqwkasskuwepwqtwvspgoelkcq
wgoviokhkazkqkxzgyhcecmeiujoqkwfwvefqhkijrclrl
```

```
twlauqrhwdmwlgusgikkflryvcwvspgpmlkassjvoqxegg
ikvrdrygfrjljslveggveyggeiapuuisfpbtgnwwmuczrv
```

Find the frequencies of the letters in the ciphertext:

```
> frequency(vvhq)

 [ 8, 5, 12, 4, 15, 10, 27, 16, 13,
14, 17, 25, 7, 7, 5, 9, 14, 17,
              24, 8, 12, 22, 22, 5,
8, 5]
```

Let’s compute the coincidences for displacements of 1, 2, 3, 4, 5, 6:

```
> coinc(vvhq,1)
                14
> coinc(vvhq,2)
                14
> coinc(vvhq,3)
                16
> coinc(vvhq,4)
                14
```

```
> coinc(vvhq,5)
                24
> coinc(vvhq,6)
                12
```

We conclude that the key length is probably 5. Let's look at the 1st, 6th, 11th, ... letters (namely, the letters in positions congruent to 1 mod 5):

```
> choose(vvhq, 5, 1)

"vvuttcccqgcunjtpjgkuqpknjkygkkgcjfqrkqjrquduk

> frequency(%)

[0, 0, 7, 1, 1, 2, 9, 0, 1, 8, 8,
0, 0, 3, 0, 4, 5, 2, 0, 3, 6, 5, 1,
0, 1, 0]
```

To express this as a vector of frequencies:

```
> vigvec(vvhq, 5, 1)

[0., 0., .1044776119, .01492537313,
.01492537313,
.02985074627, .1343283582, 0.,
.01492537313, .1194029851,
.1194029851, 0., 0., .04477611940,
0., .05970149254,
.07462686567, .02985074627, 0.,
.04477611940, .08955223881,
.07462686567, .01492537313, 0.,
.01492537313, 0.]
```

The dot products of this vector with the displacements of the alphabet frequency vector are computed as follows:

```
> corr(%)

  .02501492539, .03910447762,
.07132835821, .03882089552,
  .02749253732, .03801492538,
.05120895523, .03014925374,
  .03247761194, .04302985074,
.03377611940, .02985074628,
  .03426865672, .04456716420,
```

```
.03555223882, .04022388058,
  .04343283582, .05017910450,
.03917910447, .02958208957,
  .03262686569, .03917910448,
.03655223881, .03161194031,
  .04883582088, .03494029848
```

The third entry is the maximum, but sometimes the largest entry is hard to locate. One way to find it is

```
> max(%)

                .07132835821
```

Now it is easy to look through the list and find this number (it usually occurs only once). Since it occurs in the third position, the first shift for this Vigenère cipher is by 2, corresponding to the letter c. A procedure similar to the one just used (using `vigvec(vvhq, 5,2),...,`

`vigvec(vvhq,5,5)`) shows that the other shifts are probably 14, 3, 4, 18. Let's check that we have the correct key by decrypting.

```
> vigenere(vvhq, -[2, 14, 3, 4,
18])

themethodusedforthepreparationandreadingofcode
extremeandatthesametimeimpossibleoftranslation
eeasewithwhichthekeymaybechangedisanotherpoint
fthiscodebythosedesiringtotransmitimportantmes
estdangeroftheirmessagesbeingreadbypoliticalor
```

For the record, the plaintext was originally encrypted by the command

```
> vigenere(%, [2, 14, 3, 4, 18])

vvhqwvvrhmusgjgthkihtssejchlsfcbgvwcrlryqtfsvg
hbltspisprdxljsveeghlqwkasskuwepwqtwvspgoelkcq
wgoviokhkazkqkxzgyhcecmeiujoqkwfwvefqhkijrclrl
twlauqrhwdmwlgusgikkflryvcwvspgpmlkassjvoqxegg
```

```
ikvrdrygfrjljslveggveyggeiapuuisfpbtgnwwmuczrv
```

B.4 Examples for Chapter 3

Example 5

Find gcd(23456, 987654).

```
> gcd(23456, 987654)

                    2
```

Example 6

Solve 23456x+987654y=2 in integers x, y.

```
igcdex(23456, 987654,'x','y')

                        2
```

```
> x;y

                              -3158
                               75
```

This means that 2 is the gcd and 23456·(−3158)+987654·75=2. (The command `igcdex` is for *integer gcd extended*. Maple also calculates gcd's for polynomials.) Variable names other than `'x'` and `'y'` can be used if these letters are going to be used elsewhere, for example, in a polynomial. We can also clear the value of `x` as follows:

```
> x:='x'

                          x:=x
```

Example 7

Compute 234·456 (mod 789).

```
> 234*456 mod 789

                              189
```

Example 8

Compute 234567876543 (mod 565656565).

```
> 234567&a^876543 mod 565656565

                         473011223
```

You might need a \ before the a^. Use the right arrow to escape from the exponent mode.

Example 9

Find the multiplicative inverse of $87878787 \pmod{9191919191}$.

```
> 87878787&a^(-1) mod 9191919191

                    7079995354
```

You might need a space before the exponent (-1). (The command `1/87878787 mod 9191919191` also works).

Example 10

Solve $7654x \equiv 2389 \pmod{65537}$.

SOLUTION

Here is one way.

```
> solve(7654*x=2389,x) mod 65537

                          43626
```

Here is another way.

```
> 2389/7654 mod 65537

                          43626
```

The fraction `2389/7654` will appear as a vertically set fraction 23897654. Use the right arrow key to escape from the fraction mode.

Example 11

Find x with

$x \equiv 2 \pmod{78}$, $x \equiv 5 \pmod{97}$, $x \equiv 1 \pmod{119}$.

```
> chrem([2, 5, 1],[78, 97, 119])
```

```
                    647480
```

We can check the answer:

```
> 647480 mod 78; 647480 mod 97;
647480 mod 119

                        2
                        5
                        1
```

Example 12

Factor 123450 into primes.

```
> ifactor(123450)

                  (2)(3)(5)2(823)
```

This means that 123450=2131528231.

Example 13

Evaluate $\phi(12345)$.

```
> phi(12345)

                6576
```

Example 14

Find a primitive root for the prime 65537.

```
> primroot(65537)

                   3
```

Therefore, 3 is a primitive root for 65537.

Example 15

Find the inverse of the matrix $1312354153627168^{10} \pmod{999}$.

SOLUTION

First, invert the matrix without the mod, and then reduce the matrix mod 999:

```
> inverse(matrix(3,3,[13, 12, 35,
41, 53, 62, 71, 68, 10]))
```

[
368634139−226034139111134139−399234139235534139−629341399753413
]

```
> map(x->x mod 999, %)
```

[772472965641516851150133149]

This is the inverse matrix mod 999.

Example 16

Find a square root of 26951623672 mod the prime p=98573007539.

SOLUTION

Since $p \equiv 3 \pmod 4$, we can use the proposition of Section 3.9:

```
> 26951623672&a^((98573007539 +
1)/4) mod 98573007539

                    98338017685
```

(You need two right arrows to escape from the fraction mode and then the exponent mode.) The other square root is minus the preceding one:

```
> -% mod 98573007539
```

```
                    234989854
```

Example 17

Let n=34222273=9803·3491. Find all four solutions of x2 ≡ 19101358 (mod 34222273).

SOLUTION

First, find a square root mod each of the two prime factors, both of which are congruent to 3 (mod 4):

```
> 19101358&a^((9803 + 1)/4) mod
9803

                      3998

> 19101358&a^((3491 + 1)/4) mod
```

```
3491

                        1318
```

Therefore, the square roots are congruent to ±3998 (mod 9803) and are congruent to ±1318 (mod 3491). There are four ways to combine these using the Chinese remainder theorem:

```
> chrem([3998, 1318],[9803, 3491])

                      43210

> chrem([-3998, 1318],[9803, 3491])

                     8397173

> chrem([3998, -1318],[9803, 3491])

                    25825100

> chrem([-3998, -1318],[9803,
3491])
```

These are the four desired square roots.

B.5 Examples for Chapter 5

Example 18

Compute the first 50 terms of the recurrence

$$x_{n+5} \equiv x_n + x_{n+2} \pmod 2.$$

The initial values are 0, 1, 0, 0, 0.

SOLUTION

The vector of coefficients is [1, 0, 1, 0, 0] and the initial values are given by the vector [0, 1, 0, 0, 0]. Type

```
> lfsr([1, 0, 1, 0, 0], [0, 1, 0,
0, 0], 50)
```

```
[0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0,
1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0,
1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0,
0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1,
1, 1, 1]
```

Example 19

Suppose the first 20 terms of an LFSR sequence are 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1. Find a recurrence that generates this sequence.

SOLUTION

First, we need to find the length of the recurrence. The command `lfsrlength(v, n)` calculates the determinants mod 2 of the first n matrices that appear in the procedure in Section 5.2:

```
> lfsrlength([1, 0, 1, 0, 1, 1, 1,
0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0,
1], 10)

                        [1, 1]
                        [2, 1]
                        [3, 0]
                        [4, 1]
                        [5, 0]
                        [6, 1]
                        [7, 0]
                        [8, 0]
                        [9, 0]
                        [10, 0]
```

The last nonzero determinant is the sixth one, so we guess that the recurrence has length 6. To find the coefficients:

```
> lfsrsolve([1, 0, 1, 0, 1, 1, 1,
0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0,
1], 6)

                  [1, 0, 1, 1, 1, 0]
```

This gives the recurrence as

$$x_{n+6} \equiv x_n + x_{n+2} + x_{n+3} + x_{n+4} \pmod 2.$$

Example 20

The ciphertext 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0 was produced by adding the output of a LFSR onto the plaintext mod 2 (i.e., XOR the plaintext with the LFSR output). Suppose you know that the plaintext starts 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0. Find the rest of the plaintext.

SOLUTION

XOR the ciphertext with the known part of the plaintext to obtain the beginning

of the LFSR output:

```
> [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
0, 1, 1, 1, 0, 0]
  + [0, 1, 1, 0, 1, 0, 1, 0, 1, 0,
0, 1, 1, 0, 0, 0, 1] mod 2

       [1, 0, 0, 1, 0, 1, 1, 0, 1,
0, 0, 1, 0, 1, 1, 0, 1]
```

This is the beginning of the LFSR output. Now let's find the length of the recurrence.

```
> lfsrlength(%, 8)

                        [1, 1]
                        [2, 0]
                        [3, 1]
                        [4, 0]
                        [5, 1]
                        [6, 0]
                        [7, 0]
                        [8, 0]
```

We guess the length is 5. To find the coefficients of the recurrence:

```
> lfsrsolve(%%, 5)

                  [1, 1, 0, 0, 1]
```

Now we can generate the full output of the LFSR using the coefficients we just found plus the first five terms of the LFSR output:

```
> lfsr([1, 1, 0, 0, 1], [1, 0, 0,
1, 0], 40)

[1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0,
1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1,
0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0,
1, 0, 1, 1, 0]
```

When we XOR the LFSR output with the ciphertext, we get back the plaintext:

```
> % + [0, 1, 1, 0, 1, 0, 1, 0, 1,
0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1,
0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0,
0, 0, 1, 0, 1, 1, 0] mod 2

[1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0,
0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0,
0, 0, 0, 0, 0]
```

This is the plaintext.

B.6 Examples for Chapter 6

Example 21

The ciphertext

22, 09, 00, 12, 03, 01, 10, 03, 04, 08, 01, 17

was encrypted using a Hill cipher with matrix

1234567810.

Decrypt it.

SOLUTION

There are several ways to input a matrix. One way is the following. A 2×2 matrix abcd can be entered as `matrix(2,2,`

`[a,b,c,d])`. Type `evalm(M&*N)` to multiply matrices M and N. Type `evalm(v&*M)` to multiply a vector v on the right by a matrix M.

Here is the encryption matrix.

```
> M:=matrix(3,3,
[1,2,3,4,5,6,7,8,10])
```

[1234567810]

We need to invert the matrix mod 26:

```
> invM:=map(x->x mod 26,
inverse(M))
```

[8161821241241]

The command `map(x->x mod 26, E)` takes each number in an expression `E` and reduces it mod 26.

This is the inverse of the matrix mod 26. We can check this as follows:

```
> evalm(M&*invM)
```

[271305278313130130520209]

```
> map(x->x mod 26, %)
```

[100010001]

To decrypt, we break the ciphertext into blocks of three numbers and multiply each block on the right by the inverse matrix we just calculated:

```
> map(x->x mod 26,
evalm([22,09,00]&*invM))

                    [14, 21, 4]

> map(x->x mod 26,
evalm([12,03,01]&*invM))
```

```
                        [17, 19, 7]

> map(x->x mod 26,
evalm([10,03,04]&*invM))

                         [4, 7, 8]

> map(x->x mod 26,
evalm([08,01,17]&*invM))

                        [11, 11, 23]
```

Therefore, the plaintext is 14, 21, 4, 17, 19, 7, 4, 7, 8, 11, 11, 23. Changing this back to letters, we obtain `overthehillx`. Note that the final `x` was appended to the plaintext in order to complete a block of three letters.

B.7 Examples for Chapter 9

Example 22

Suppose you need to find a large random prime of 50 digits. Here is one way. The function `nextprime` finds the next prime greater than x. The function `rand(a..b)()` gives a random integer between a and b. Combining these, we can find a prime:

```
> nextprime(rand(10^49..10^50)())

73050570031667109175215303340488313456708913288
```

If we repeat this procedure, we should

get another prime:

```
> nextprime(rand(10^49..10^50)())

9747640769493130325572432604058614414534105456
```

Example 23

Suppose you want to change the text *hellohowareyou* to numbers:

```
> text2num("hellohowareyou")

805121215081523011805251521
```

Note that we are now using $a = 1$, $b = 2$, ..., $z = 26$, since otherwise *a*'s at the beginnings of messages would disappear. (A more efficient procedure

would be to work in base 27, so the numerical form of the message would be $8+5\cdot 27+12\cdot 272+\cdots +21\cdot 2713=87495221502384554951$. Note that this uses fewer digits.)

Now suppose you want to change it back to letters:

```
> num2text(805121215081523011805251521)

                    "hellohowareyou"
```

Example 24

Encrypt the message `hi` using RSA with $n=823091$ and $e=17$.

SOLUTION

First, change the message to numbers:

```
> text2num("hi")

                809
```

Now, raise it to the eth power mod n:

```
> %&a^17 mod 823091

                     596912
```

You might need a \ before the a^. Use the right arrow to escape from the exponent mode.

Example 25

Decrypt the ciphertext in the previous problem.

SOLUTION

First, we need to find the decryption exponent d. To do this, we need to find $\phi(823091)$. One way is

```
> phi(823091)

                821184
```

Another way is to factor n as $p \cdot q$ and then compute $(p-1)(q-1)$:

```
> ifactor(823091)

                (659)(1249)

> 658*1248

                  821184
```

Since $de \equiv 1 \pmod{\phi(n)}$, we compute

the following (note that we are finding the inverse of e mod $\phi(n)$, not mod n):

```
> 17&a^ (-1) mod 821184

                        48305
```

Therefore, d=48305. To decrypt, raise the ciphertext to the dth power mod n:

```
> 596912&a^48305 mod 823091
                         809
```

Finally, change back to letters:

```
> num2text(809)

                    "hi"
```

Example 26

Encrypt `hellohowareyou` using RSA with n=823091 and e=17.

SOLUTION

First, change the plaintext to numbers:

```
> text2num("hellohowareyou")

805121215081523011805251521
```

Suppose we simply raised this to the eth power mod n:

```
> %&a^17 mod 823091

                    447613
```

If we decrypt (we know d from Example

25), we obtain

```
> %&a^48305 mod 823091

                    628883
```

This is not the original plaintext. The reason is that the plaintext is larger than n, so we have obtained the plaintext mod n:

```
> 805121215081523011805251521 mod
823091

                        628883
```

We need to break the plaintext into blocks, each less than n. In our case, we use three letters at a time:

```
805121215081523011805251521
```

```
> 80512&a^17 mod 823091

                    757396

> 121508&a^17 mod 823091

                    164513

> 152301&a^17 mod 823091

                    121217

> 180525&a^17 mod 823091

                    594220

> 1521&a^17 mod 823091

                    442163
```

The ciphertext is therefore `757396164513121217594220442163`. Note that there is no reason to change this back to letters. In fact, it doesn't correspond to any text with letters.

Decrypt each block individually:

```
> 757396&a^48305 mod 823091

                        80512

> 164513&a^48305 mod 823091

                        121508
```

etc.

We'll now do some examples with large numbers, namely the numbers in the RSA Challenge discussed in Section 9.5. These are stored under the names *rsan, rsae, rsap, rsaq*:

```
> rsan

114381625757888867669235779976146612010218296
706935245733897830597123563958705058989075147
```

```
> rsae
                    9007
```

Example 27

Encrypt each of the messages `b, ba, bar, bard` using *rsan* and *rsae*.

```
> text2num("b")&a^rsae mod rsan

709467584676126685983701649915507861828763310 6
867822617164972001221553323484620140532879875 8

> text2num("ba")&a^rsae mod rsan

350451306089751003250117094498719542737882047 5
276217598060279622705380315655647733520336717 8

> text2num("bar")&a^rsae mod rsan

448145128638551010760045308594921093424295316 0
008436459868804059531028183128225863625802987 8
```

```
> text2num("bard")&a^rsae mod rsan

242380777851116664232028625120903173934852129556054323297179804928958073445752663026449873980
```

Observe that the ciphertexts are all the same length. There seems to be no easy way to determine the length of the corresponding plaintext.

Example 28

Using the factorization rsap=rsap·rsap, find the decryption exponent for the RSA Challenge, and decrypt the ciphertext (see Section 9.5).

First we find the decryption exponent:

```
> rsad:=rsae&a^(-1) mod
(rsap-1)*(rsaq-1):
```

Note that we use the final colon to avoid printing out the value. If you want to see the value of *rsad*, see Section 9.5, or don't use the colon. To decrypt the ciphertext, which is stored as *rsaci*, and change to letters:

```
> num2text(rsaci&a^rsad mod rsan)

        "the magic words are
squeamish ossifrage"
```

Example 29

Encrypt the message `rsaencryptsmessageswell` using *rsan* and *rsae*.

```
>
```

```
text2num("rsaencryptsmessageswell")&a^rsae
mod rsan

946394203490022593163058235392494964146409693
719506542543655849060139663288177535392831126
```

Example 30

Decrypt the preceding ciphertext.

SOLUTION

Fortunately, we know the decryption exponent *rsad*. Therefore, we compute

```
> %&a^ rsad mod rsan

1819010514031825162019130519190107051923051212

> num2text(%)
```

```
"rsaencryptsmessageswell"
```

Suppose we lose the final digit 4 of the ciphertext in transmission. Let's try to decrypt what's left (subtracting 4 and dividing by 10 is a mathematical way to remove the 4):

```
> (%%% - 4)/10)&a^rsad mod rsan

47952999173195988664902352629525486409113633889
88412300373487969657794254117158956921267912626
```

If we try to change this to letters, we do not get anything resembling the message. A small error in the plaintext completely changes the decrypted message and usually produces garbage.

Example 31

Suppose we are told that n=11313771275590312567 is the product of two primes and that ϕ(n)=11313771187608744400. Factor n.

SOLUTION

We know (see Section 9.1) that p and q are the roots of X2−(n−ϕ(n)+1)X+n. Therefore, we compute

```
> solve(xa^2 -
(11313771275590312567 -
11313771187608744400 + 1)*x +
11313771275590312567, x)

                87852787151,
128781017
```

Therefore, n=128781017·87852787151.

We also could have used the quadratic formula to find the roots.

Example 32

Suppose we know *rsae* and *rsad*. Use these to factor *rsan*.

SOLUTION

We use the ar ≡ 1 factorization method from Section 9.4. First write rsae·rsad −1=2sm with m odd. One way to do this is first to compute rsae·rsad−1, and then keep dividing by 2 until we get an odd number:

```
> rsae*rsad - 1

961034419617782266156919023359583834109854129015598557508735265915617489855734299513159468040
```

```
> %/2

480517209808891133078459511679791917054927064
577992787543676329578087449278671497565797340

> %/2
240258604904445566539229755839895958527463532
288996393771838164789043724639335748782898670
```

We continue this way for six more steps until we get

```
375404070163196197717546493499837435199161769
576556865268497132482880819748962107473279172
```

This number is m. Now choose a random integer a. Hoping to be lucky, we choose 13. As in the ar ≡ 1 factorization method, we compute

```
> 13&a^% mod rsan
```

```
2757436850700653059224349486884716119842309570
3018310983986237080052933809298479549019264358
```

Since this is not ±1 (mod rsan), we successively square it until we get ±1:

```
> %&a^2 mod rsan

4831896032192851558013847641872303455410409906
5499641258295563603526615610868643119429857407

> %&a^2 mod rsan

7817281415487735657914192805875400002194878705
1518183974205601327552191348756094473207351648

> %&a^2 mod rsan

4283619120250872874219929904058290020297622291
0944451823946218637947056944205510139299229308

> %&a^2 mod rsan

                    1
```

Since the last number before the 1 was

not $\pm 1 \pmod{\text{rsan}}$, we have an example of $x \equiv \pm 1 \pmod{\text{rsan}}$ with $x2 \equiv 1$. Therefore, gcd(x−1, rsan) is a nontrivial factor of *rsan*:

```
> gcd(%% - 1, rsan)

3276913299326670954996198819083446141317764296
```

This is *rsaq*. The other factor is obtained by computing *rsan/rsaq*:

```
rsan/%

3490529510847650949147849619903898133417764638
```

This is *rsap*.

Example 33

Suppose you know that

$1508834755694512 \equiv 168875705328582 \pmod{205611444308117}$.

Factor 205611444308117.

SOLUTION

We use the Basic Principle of Section 9.4:

```
> gcd(150883475569451-16887570532858,2056114443
                        23495881
```

This gives one factor. The other is

```
> 205611444308117/%
                    8750957
```

We can check that these factors are

actually primes, so we can't factor any further:

```
> isprime(%%)

                true

> isprime(%%)

                true
```

Example 34

Factor n=376875575426394855599989992897873239 by the p−1 method.

SOLUTION

Let's choose our bound as B=100, and let's take a=2, so we compute 2100!

(mod n):

```
> 2&a^factorial(100)
mod
3768755754263948555999899928978732 39


369676678301956331939422106251199512
```

Then we compute the gcd of 2100!−1 and n:

```
> gcd(% - 1,
3768755754263948555999899928978732 39)

                    430553161739796481
```

This is a factor p. The other factor q is

```
>
3768755754263948555999899928978732 39/
%
```

```
                    875328783798732119
```

Let's see why this worked. The factorizations of p−1 and q−1 are

```
> ifactor(430553161739796481 - 1)

                (2) 18(3) 7(5)(7)
4(11) 3(47)

> ifactor(875328783798732119 - 1)

              (2)(61)(8967967)
(20357)(39301)
```

We see that 100! is a multiple of p−1, so $2^{100!} \equiv 1 \pmod{p}$. However, 100! is not a multiple of q−1, so it is likely that $2^{100!} \not\equiv 1 \pmod{q}$. Therefore, both $2^{100!}-1$ and pq have p as a factor, but only pq has q as a factor. It follows that the gcd is p.

B.8 Examples for Chapter 10

Example 35

Let’s solve the discrete log problem $2x \equiv 71 \pmod{131}$ by the Baby Step-Giant Step method of Subsection 10.2.2. We take N=12 since N2>p−1=130 and we form two lists. The first is $2j \pmod{131}$ for $0 \leq j \leq 11$:

```
>for j from 0 while j <= 11 do; (j,
2&j mod 131); end do;

                        0, 1
                        1, 2
                        2, 4
                        3, 8
```

```
4, 16
5, 32
6, 64
7, 128
8, 125
9, 119
10, 107
11, 83
```

The second is $71 \cdot 2^{-j} \pmod{131}$ for $0 \le j \le 11$:

```
> for j from 0 while j <= 11 do;
(j, 71*2&\^\: (-12*j) mod 131); end
do;
```

```
0, 71
1, 17
2, 124
3, 26
4, 128
5, 86
6, 111
7, 93
8, 85
9, 96
```

```
10, 130
11, 116
```

The number 128 is on both lists, so we see that $27 \equiv 71 \cdot 2 - 12 * 4 \pmod{131}$. Therefore,

$$71 \equiv 27 + 4 * 12 \equiv 255 \pmod{131}.$$

B.9 Examples for Chapter 12

Example 36

Suppose there are 23 people in a room. What is the probability that at least two have the same birthday?

SOLUTION

The probability that no two have the same birthday is $\prod i=122(1-i/365)$ (note that the product stops at i=22, not i=23). Subtracting from 1 gives the probability that at least two have the same birthday:

```
> 1-mul(1.-i/365, i=1..22)

                                .5072972344
```

Note that we used 1. in the product instead of 1 without the decimal point. If we had omitted the decimal point, the product would have been evaluated as a rational number (try it, you'll see).

Example 37

Suppose a lazy phone company employee assigns telephone numbers by choosing random seven-digit numbers. In a town with 10,000 phones, what is the probability that two people receive the same number?

```
> 1-mul(1.-i/10a^7, i=1..9999)
```

```
                                .9932699133
```

Note that the number of phones is about three times the square root of the number of possibilities. This means that we expect the probability to be high, which it is. From Section 12.1, we have the estimate that if there are around $2(\ln 2)107 \approx 3723$ phones, there should be a 50% chance of a match. Let's see how accurate this is:

```
> 1-mul(1.-i/10a^7, i=1..3722)

                                .4998945441
```

B.10 Examples for Chapter 17

Example 38

Suppose we have a (5, 8) Shamir secret sharing scheme. Everything is mod the prime p=987541. Five of the shares are

(9853, 853), (4421, 4387), (6543, 1234), (93293, 78428), (12398, 7563).

Find the secret.

SOLUTION

One way: First, find the Lagrange interpolating polynomial through the five points:

```
>
interp([9853,4421,6543,93293,12398],
[853,4387,1234,78428,7563],x)
```

```
-4959003720134640533754713378864151099487659484
+2044843261540449832301145924339442825912229810
```

Now evaluate at x=0 to find the constant term:

```
> eval(%,x=0)
```

```
204484326154044983230114592433944282591222981 0
```

We need to change this to an integer mod 987541:

```
> % mod 987541
```

```
678987
```

Therefore, 678987 is the secret.

Here is another way. Set up the matrix equations as in the text and then solve for the coefficients of the polynomial mod 987541:

```
> map(x->x mod
987541,evalm(inverse(matrix(5,5,
[1,9853,9853a^2,9853a^3,9853a^4,
1,4421,4421a^2,4421a^3,4421a^4,
1,6543,6543a^2,6543a^3, 6543a^4,
1, 93293, 93293a^2,93293a^3,
93293a^4,
1, 12398,
12398a^2,12398a^3,12398a^4]))
&*matrix(5,1,
[853,4387,1234,78428,7563])))
```

```
[ 678987147281651574413456741 ]
```

The constant term is 678987, which is the secret.

B.11 Examples for Chapter 18

Example 39

Here is a game you can play. It is essentially the simplified version of poker over the telephone from Section 18.2. There are five cards: ten, jack, queen, king, ace. They are shuffled and disguised by raising their numbers to a random exponent mod the prime `24691313099`. You are supposed to guess which one is the ace.

To start, pick a random exponent. We use the colon after `khide()` so that we cannot cheat and see what value of k is

being used.

```
> k:= khide():
```

Now, shuffle the disguised cards (their numbers are raised to the kth power mod p and then randomly permuted):

```
> shuffle(k)

  [14001090567, 16098641856,
23340023892, 20919427041,
7768690848]
```

These are the five cards. None looks like the ace that's because their numbers have been raised to powers mod the prime. Make a guess anyway. Let's see if you're correct.

```
> reveal(%)

          ["ten", "ace", "queen",
"jack", "king"]
```

Let’s play again:

```
> k:= khide():

> shuffle(k)

  [13015921305, 14788966861,
23855418969, 22566749952,
8361552666]
```

Make your guess (note that the numbers are different because a different random exponent was used). Were you lucky?

```
> reveal(%)

             ["ten", "queen", "ace",
"king", "jack"]
```

Perhaps you need some help. Let's play one more time:

```
> k:= khide():
> shuffle(k)

  [13471751030, 20108480083,
8636729758, 14735216549,
11884022059]
```

We now ask for advice:

```
> advise(%)

             3
```

We are advised that the third card is the ace. Let's see (recall that %% is used to refer to the next to last output):

```
> reveal(%%)
```

```
            ["jack", "ten", "ace",
"queen", "king"]
```

How does this work? Read the part on "How to Cheat" in Section 18.2. Note that if we raise the numbers for the cards to the (p−1)/2 power mod p, we get

```
> map(x->x&a^((24691313099-1)/2)
mod 24691313099,
[200514, 10010311, 1721050514,
11091407, 10305])

                [1, 1, 1, 1,
24691313098]
```

Therefore, only the ace is a quadratic nonresidue mod p.

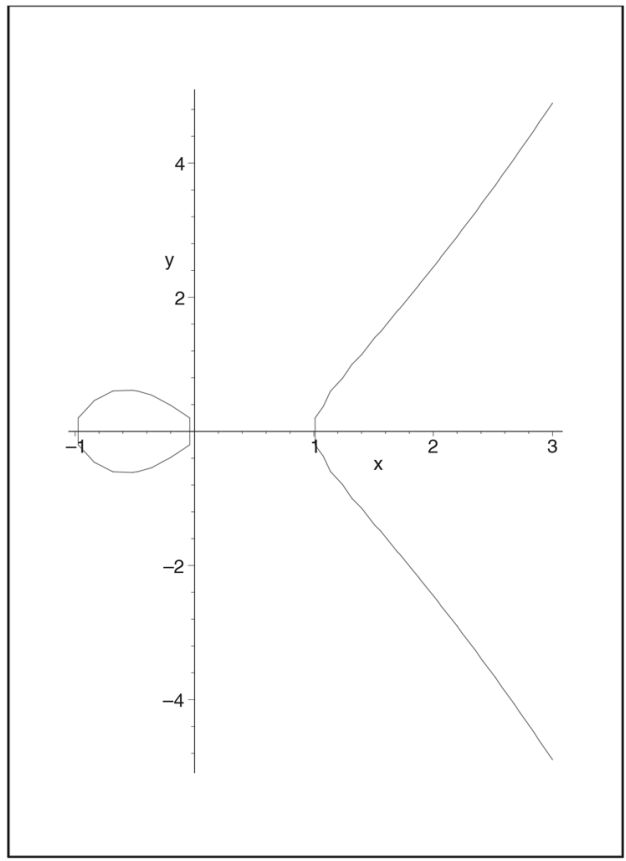
B.12 Examples for Chapter 21

Example 40

All of the elliptic curves we work with in this chapter are elliptic curves mod n. However, it is helpful use the graphs of elliptic curves with real numbers in order to visualize what is happening with the addition law, for example, even though such pictures do not exist mod n.

Let's graph the elliptic curve $y2=x(x-1)(x+1)$. We'll specify that $-1 \le x \le 3$ and $-5 \le y \le 5$, and make sure that `x` and `y` are cleared of previous values.

```
>
x:='x';y:='y';implicitplot(ya^2=x*(x-1)*(x
+1), x=-1..3,y=-5..5)
```



B.12-1 Full Alternative Text

Example 41

Add the points (1, 3) and (3, 5) on the

elliptic curve $y2 \equiv x3+24x+13 \pmod{29}$.

```
> addell([1,3], [3,5], 24, 13, 29)

                          [26,1]
```

You can check that the point (26, 1) is on the curve: $263+24 \cdot 26+13 \equiv 12 \pmod{29}$.

Example 42

Add (1, 3) to the point at infinity on the curve of the previous example.

```
> addell([1,3],
["infinity","infinity" ], 24, 13,
29)

                          [1,3]
```

As expected, adding the point at infinity to a point P returns the point P.

Example 43

Let P=(1, 3) be a point on the elliptic curve y2 ≡ x3+24x+13 (mod 29). Find 7P.

```
> multell([1,3], 7, 24, 13, 29)

                          [15,6]
```

Example 44

Find k(1, 3) for k=1, 2, 3, … , 40 on the curve of the previous example.

```
> multsell([1,3], 40, 24, 13, 29)
```

```
[[1,[1,3]],[2,[11,10]],[3,[23,28]],
[4,[0,10]],[5,[19,7]],[6,[18,19]],
[7,[15,6]],[8,[20,24]],[9,[4,12]],
[10,[4,17]],[11,[20,5]],
[12,[15,23]],[13,[18,10]],[14,
[19,22]],[15,[0,19]],[16,[23,1]],
[17,[11,19]],[18,[1,26]],[19,
["infinity","infinity"]],[20,
[1,3]],
[21,[11,10]],[22,[23,28]],[23,
[0,10]],[24,[19,7]],[25,[18,19]],
[26,[15,6]],[27,[20,24]],[28,
[4,12]],[29,[4,17]],[30,[20,5]],
[31,[15,23]],[32,[18,10]],[33,
[19,22]],[34,[0,19]],[35,[23,1]],
[36,[11,19]],[37,[1,26]],[38,
["infinity","infinity"]],[39,
[1,3]],
[40,[11,10]]]
```


Notice how the points repeat after every 19 multiples.

Example 45

The previous four examples worked mod the prime 29. If we work mod a composite number, the situation at infinity becomes more complicated since we could be at infinity mod both factors or we could be at infinity mod one of the factors but not mod the other. Therefore, we stop the calculation if this last situation happens and we exhibit a factor. For example, let's try to compute 12P, where P=(1, 3) is on the elliptic curve $y2 \equiv x3-5x+13 \pmod{209}$:

```
> multell([1,3], 12, -5, 13, 11*19)

                    ["factor=",19]
```

Now let's compute the successive multiples to see what happened along the way:

```
> multsell([1,3], 12, -5, 13,
11*19)

[[1,[1,3]],[2,[91,27]],[3,
[118,133]],[4,[148,182]],[5,
[20,35]],
                [6,["factor=",19]]]
```

When we computed 6P, we ended up at infinity mod 19. Let's see what is happening mod the two prime factors of 209, namely 19 and 11:

```
> multsell([1,3], 12, -5, 13, 19)

[[1,[1,3]],[2,[15,8]],[3,[4,0]],[4,
[15,11]],[5,[1,16]],
[6,["infinity","infinity"]],[7,
[1,3]],[8,[15,8]],[9,[4,0]],
[10,[15,11]],[11,[1,16]],[12,
["infinity","infinity"]]]

> multsell([1,3], 24, -5, 13, 11)
```

```
[[1,[1,3]],[2,[3,5]],[3,[8,1]],[4,
[5,6]],[5,[9,2]],[6,[6,10]],
[7,[2,0]],[8,[6,1]],[9,[9,9]],[10,
[5,5]],[11,[8,10]],[12,[3,6]],
[13,[1,8]],[14,
["infinity","infinity"]],[15,
[1,3]],[16,[3,5]],
[17,[8,1]],[18,[5, 6]],[19,[9, 2]],
[20,[6,10]],[21,[2,0]],
[22,[6,1]],[23,[9,9]],[24,[5,5]]]
```

After six steps, we were at infinity mod 19, but it takes 14 steps to reach infinity mod 11. To find 6P, we needed to invert a number that was 0 mod 19 and nonzero mod 11. This couldn't be done, but it yielded the factor 19. This is the basis of the elliptic curve factorization method.

Example 46

Factor 193279 using elliptic curves.

SOLUTION

First, we need to choose some random elliptic curves and a point on each curve. For example, let's take P=(2, 4) and the elliptic curve

$$y2 \equiv x3-10x+b \pmod{193279}.$$

For P to lie on the curve, we take b=28. We'll also take

$$y2 \equiv x3+11x-11,\ P=(1, 1) y2 \equiv x3+17x-14,\ P=(1, 2).$$

Now we compute multiples of the point P. We do the analog of the p−1 method, so we choose a bound B, say B=12, and compute B!P.

```
> multell([2,4], factorial(12),
-10, 28, 193279)
```

```
                    ["factor=",347]

> multell([1,1], factorial(12), 11,
-11, 193279)

                     [13862,35249]

> multell([1,2], factorial(12), 17,
-14, 193279)

                    ["factor=",557]
```

Let's analyze in more detail what happened in these examples.

On the first curve, 266P ends up at infinity mod 557 and 35P is infinity mod 347. Since 266=2·7·19, it has a prime factor larger than B=12, so B!P is not infinity mod 557. But 35 divides B!, so B!P is infinity mod 347.

On the second curve, 356P=infinity mod

347 and 561P=infinity mod 557. Since 356=4·89 and 561=3·11·17, we don't expect to find the factorization with this curve.

The third curve is a surprise. We have 331P=infinity mod 347 and 272P=infinity mod 557. Since 331 is prime and 272=16·17, we don't expect to find the factorization with this curve. However, by chance, an intermediate step in the calculation of B!P yielded the factorization. Here's what happened. At one step, the program required adding the points (184993, 13462) and (20678, 150484). These two points are congruent mod 557 but not mod 347. Therefore, the slope of the line through these two points is defined mod 347 but

is 0/0 mod 557. When we tried to find the multiplicative inverse of the denominator mod 193279, the gcd algorithm yielded the factor 557. This phenomenon is fairly rare.

Example 47

Here is how to produce the example of an elliptic curve ElGamal cryptosystem from Section 21.5. For more details, see the text. The elliptic curve is $y2 \equiv x3+3x+45 \pmod{8831}$ and the point is G=(4, 11). Alice's message is the point Pm=(5, 1743).

Bob has chosen his secret random number aB=3 and has computed abG:

```
> multell([4,11], 3, 3, 45, 8831)

                [413,1808]
```

Bob publishes this point. Alice chooses the random number k=8 and computes kG and Pm+k(aBG):

```
> multell([4,11], 8, 3, 45, 8831)

                    [5415,6321]

>
addell([5,1743],multell([413,1808],8,3,45,8831

                    [6626,3576]
```

Alice sends (5415,6321) and (6626,3576) to Bob, who multiplies the first of these point by aB:

```
> multell([5415,6321], 3, 3, 45,
8831)
```

```
                    [673,146]
```

Bob then subtracts the result from the last point Alice sends him. Note that he subtracts by adding the point with the second coordinate negated:

```
> addell([6626,3576], [673,-146],
3, 45, 8831)

                    [5,1743]
```

Bob has therefore received Alice's message.

Example 48

Let's reproduce the numbers in the example of a Diffie-Hellman key exchange from Section 21.5: The elliptic

curve is y2 ≡ x3+x+7206 (mod 7211) and the point is G=(3, 5). Alice chooses her secret NA=12 and Bob chooses his secret NB=23. Alice calculates

```
> multell([3,5], 12, 1, 7206, 7211)

                    [1794,6375]
```

She sends (1794,6375) to Bob. Meanwhile, Bob calculates

```
> multell([3,5], 23, 1, 7206, 7211)

                    [3861, 1242]
```

and sends (3861,1242) to Alice. Alice multiplies what she receives by NA and Bob multiplies what he receives by NB:

```
> multell([3861,1242], 12, 1, 7206,
7211)

                    [1472,2098]

> multell([1794,6375], 23, 1, 7206,
7211)

                    [1472,2098]
```

Therefore, Alice and Bob have produced the same key.

Appendix C MATLAB® Examples

These computer examples are written for MATLAB. If you have MATLAB available, you should try some of them on your computer. For information on getting started with MATLAB, see Section C.1. Several functions have been written to allow for experimentation with MATLAB. The MATLAB functions associated with this book are available at

```
bit.ly/2HyvR8n
```

We recommend that you create a directory or folder to store these files

and download them to that directory or folder. One method for using these functions is to launch MATLAB from the directory where the files are stored, or launch MATLAB and change the current directory to where the files are stored. In some versions of MATLAB the working directory can be changed by changing the current directory on the command bar. Alternatively, one can add the path to that directory in the MATLAB path by using the *path* function or the Set Path option from the File menu on the command bar.

If MATLAB is not available, it is still possible to read the examples. They provide examples for several of the concepts presented in the book. Most of

the examples used in the MATLAB appendix are similar to the examples in the Mathematica and Maple appendices. MATLAB, however, is limited in the size of the numbers it can handle. The maximum number that MATLAB can represent in its default mode is roughly 16 digits and larger numbers are approximated. Therefore, it is necessary to use the symbolic mode in MATLAB for some of the examples used in this book.

A final note before we begin. It may be useful when doing the MATLAB exercises to change the formatting of your display. The command

```
>> format rat
```

sets the formatting to represent numbers using a fractional representation. The conventional *short* format represents large numbers in scientific notation, which often doesn't display some of the least significant digits. However, in both formats, the calculations, when not in symbolic mode, are done in floating point decimals, and then the rational format changes the answers to rational numbers approximating these decimals.

C.1 Getting Started with MATLAB

MATLAB is a programming language for performing technical computations. It is a powerful language that has become very popular and is rapidly becoming a standard instructional language for courses in mathematics, science, and engineering. MATLAB is available on most campuses, and many universities have site licenses allowing MATLAB to be installed on any machine on campus.

In order to launch MATLAB on a PC, double click on the MATLAB icon. If you want to run MATLAB on a Unix

system, type *matlab* at the prompt. Upon launching MATLAB, you will see the MATLAB prompt:

```
>>
```

which indicates that MATLAB is waiting for a command for you to type in. When you wish to quit MATLAB, type *quit* at the command prompt.

MATLAB is able to do the basic arithmetic operations such as addition, subtraction, multiplication, and division. These can be accomplished by the operators +, -, *, and /, respectively. In order to raise a number to a power, we use the operator ^ . Let us look at an example:

If we type 2^7+125/5 at the prompt and press the *Enter* key

```
>> 2^7 + 125/5
```

then MATLAB will return the answer:

```
ans =
   153
```

Notice that in this example, MATLAB performed the exponentiation first, the division next, and then added the two results. The order of operations used in MATLAB is the one that we have grown up using. We can also use parentheses to change the order in which MATLAB calculates its quantities. The following example exhibits this:

```
```

```
>> 11*( (128/(9+7) - 2^(72/12)))

ans =
   -616
```

In these examples, MATLAB has called the result of the calculations *ans*, which is a variable that is used by MATLAB to store the output of a computation. It is possible to assign the result of a computation to a specific variable. For example,

```
>> spot=17

spot =
   17
```

assigns the value of 17 to the variable *spot*. It is possible to use variables in computations:

```

```

```
>> dog=11

dog =
   11

>> cat=7

cat =
   7

>> animals=dog+cat

animals =
   18
```

MATLAB also operates like an advanced scientific calculator since it has many functions available to it. For example, we can do the standard operation of taking a square root by using the *sqrt* function, as in the following example:

```
>> sqrt(1024)
```

```
ans =
   32
```

There are many other functions available. Some functions that will be useful for this book are *mod, factorial, factor, prod*, and *size*.

Help is available in MATLAB. You may either type *help* at the prompt, or pull down the Help menu. MATLAB also provides help from the command line by typing *help commandname*. For example, to get help on the function *mod*, which we shall be using a lot, type the following:

```
>> help mod
```

MATLAB has a collection of toolboxes available. The toolboxes consist of

collections of functions that implement many application-specific tasks. For example, the Optimization toolbox provides a collection of functions that do linear and nonlinear optimization. Generally, not all toolboxes are available. However, for our purposes, this is not a problem since we will only need general MATLAB functions and have built our own functions to explore the number theory behind cryptography.

The basic data type used in MATLAB is the matrix. The MATLAB programming language has been written to use matrices and vectors as the most fundamental data type. This is natural since many mathematical and scientific problems lend themselves to using

matrices and vectors.

Let us start by giving an example of how one enters a matrix in MATLAB. Suppose we wish to enter the matrix

A=1111124813927141664

into MATLAB. To do this we type:

```
>> A = [1 1 1 1; 1 2 4 8; 1 3 9 27;
1 4 16 64]
```

at the prompt. MATLAB returns

```
A =
     1 1  1  1
     1 2  4  8
     1 3  9  27
     1 4 16  64
```

There are a few basic rules that are used

when entering matrices or vectors. First, a vector or matrix is started by using a square bracket [and ended using a square bracket]. Next, blanks or commas separate the elements of a row. A semicolon is used to end each row. Finally, we may place a semicolon at the very end to prevent MATLAB from displaying the output of the command.

To define a row vector, use blanks or commas. For example,

```
>> x = [2, 4, 6, 8, 10, 12]

x =
     2    4    6    8    10    12
```

To define a column vector, use semicolons. For example,

```
>> y=[1;3;5;7]

y =
    1
    3
    5
    7
```

In order to access a particular element of y, put the desired index in parentheses. For example, y(1) = 1, y(2) = 3, and so on.

MATLAB provides a useful notation for addressing multiple elements at the same time. For example, to access the third, fourth, and fifth elements of x, we would type

```
>> x(3:5)

ans =
     6    8    10
```

The 3:5 tells MATLAB to start at 3 and count up to 5. To access every second element of x, you can do this by

```
>> x(1:2:6)

ans =
    2    6    10
```

We may do this for the array also. For example,

```
>> A(1:2:4,2:2:4)

ans =
    1   1
    3   27
```

The notation 1:n may also be used to assign to a variable. For example,

```
>> x=1:7
```

returns

```
x =
     1     2     3     4     5     6     7
```

MATLAB provides the *size* function to determine the dimensions of a vector or matrix variable. For example, if we want the dimensions of the matrix A that we entered earlier, then we would do

```
>> size(A)

ans =
     4     4
```

It is often necessary to display numbers in different formats. MATLAB provides several output formats for displaying the result of a computation. To find a list of formats available, type

```
>> help format
```

The *short* format is the default format and is very convenient for doing many computations. However, in this book, we will be representing long whole numbers, and the *short* format will cut off some of the trailing digits in a number. For example,

```
>> a=1234567899

a =
   1.2346e+009
```

Instead of using the *short* format, we shall use the *rational* format. To switch MATLAB to using the rational format, type

```
>> format rat
```

As an example, if we do the same example as before, we now get different results:

```
>> a=1234567899

a =
   1234567899
```

This format is also useful because it allows us to represent fractions in their fractional form, for example,

```
>> 111/323

ans =
   111/323
```

In many situations, it will be convenient to suppress the results of a computation. In order to have MATLAB suppress printing out the results of a command, a

semicolon must follow the command. Also, multiple commands may be entered on the same line by separating them with commas. For example,

```
>> dogs=11, cats=7; elephants=3,
zebras=19;

dogs =
   11

elephants =
   3
```

returns the values for the variables *dogs* and *elephants* but does not display the values for *cats* and *zebras*.

MATLAB can also handle variables that are made of text. A string is treated as an array of characters. To assign a string to a variable, enclose the text with single

quotes. For example,

```
>> txt='How are you today?'
```

returns

```
txt =
   How are you today?
```

A string has size much like a vector does. For example, the size of the variable txt is given by

```
>> size(txt)
ans =
    1   18
```

It is possible to edit the characters one by one. For example, the following command changes the first word of txt:

```
>> txt(1)='W';
txt(2)='h';txt(3)='o'

txt =
   Who are you today?
```

As you work in MATLAB, it will remember the commands you have entered as well as the values of the variables you have created. To scroll through your previous commands, press the up-arrow and down-arrow. In order to see the variables you have created, type *who* at the prompt. A similar command *whos* gives the variables, their size, and their type information.

Notes. 1. To use the commands that have been written for the examples, you should run MATLAB in the directory into which you have downloaded the file

from the Web site `bit.ly/2HyvR8n`

2. Some of the examples and computer problems use long ciphertexts, etc. For convenience, these have been stored in the file `ciphertexts.m`, which can be loaded by typing *ciphertexts* at the prompt. The ciphertexts can then be referred to by their names. For example, see Computer Example 4 for Chapter 2.

C.2 Examples for Chapter 2

Example 1

A shift cipher was used to obtain the ciphertext `kddkmu`.

Decrypt it by trying all possibilities.

```
>> allshift('kddkmu')


kddkmu
leelnv
mffmow
nggnpx
ohhoqy
piiprz
qjjqsa
rkkrtb
sllsuc
tmmtvd
```

```
unnuwe
voovxf
wppwyg
xqqxzh
yrryai
zsszbj
attack
buubdl
cvvcem
dwwdfn
exxego
fyyfhp
gzzgiq
haahjr
ibbiks
jccjlt
```

As you can see, `attack` is the only word that occurs on this list, so that was the plaintext.

Example 2

Encrypt the plaintext message `cleopatra` using the affine function 7x+8:

```
>> affinecrypt('cleopatra',7,8)

ans =


'whkcjilxi'
```

Example 3

The ciphertext `mzdvezc` was encrypted using the affine function 5x+12. Decrypt it.

SOLUTION

First, solve $y \equiv 5x+12 \pmod{26}$ for x to obtain $x \equiv 5-1(y-12)$. We need to find the inverse of 5 (mod 26):

```
>> powermod(5,-1,26)

ans =
```

```
    21
```

Therefore, $x \equiv 21(y-12) \equiv 21y-12 \cdot 21$. To change $-12 \cdot 21$ to standard form:

```
>> mod(-12*21,26)

ans =
    8
```

Therefore, the decryption function is $x \equiv 21y+8$. To decrypt the message:

```
>> affinecrypt('mzdvezc',21,8)

ans =


'anthony'
```

In case you were wondering, the plaintext was encrypted as follows:

```
```

```
>> affinecrypt('anthony',5,12)

ans =


'mzdvezc'
```

Example 4

Here is the example of a Vigenère cipher from the text. Let's see how to produce the data that was used in Section 2.3 to decrypt the ciphertext. In the file ciphertexts.m, the ciphertext is stored under the name *vvhq*. If you haven't already done so, load the file ciphertexts.m:

```
>> ciphertexts
```

Now we can use the variable vvhq to

obtain the ciphertext:

```
>> vvhq

vvhqwvvrhmusgjgthkihtssejchlsfcbgvwcrlryqtfsvg
hbltspisprdxljsveeghlqwkasskuwepwqtwvspgoelkcq
wgoviokhkazkqkxzgyhcecmeiujoqkwfwvefqhkijrclrl
twlauqrhwdmwlgusgikkflryvcwvspgpmlkassjvoqxegg
ikvrdrygfrjljslveggveyggeiapuuisfpbtgnwwmuczrv
```

We now find the frequencies of the letters in the ciphertext. We use the function *frequency*. The *frequency* command was written to display automatically the letter and the count next to it. We therefore have put a semicolon at the end of the command to prevent MATLAB from displaying the count twice.

```
>> fr=frequency(vvhq);
 a   8
```

```
b   5
c  12
d   4
e  15
f  10
g  27
h  16
i  13
j  14
k  17
l  25
m   7
n   7
o   5
p   9
q  14
r  17
s  24
t   8
u  12
v  22
w  22
x   5
y   8
z   5
```

Let’s compute the coincidences for displacements of 1, 2, 3, 4, 5, 6:

```
>> coinc(vvhq,1)

ans =
   14

>> coinc(vvhq,2)

ans =
   14

>> coinc(vvhq,3)

ans =
   16

>> coinc(vvhq,4)

ans =
   14

>> coinc(vvhq,5)

ans =
   24

>> coinc(vvhq,6)

ans =
   12
```

We conclude that the key length is probably 5. Let's look at the 1st, 6th, 11th, ... letters (namely, the letters in positions congruent to 1 mod 5). The function *choose* will do this for us. The function *choose(txt,m,n)* extracts every letter from the string txt that has positions congruent to n mod m.

```
>> choose(vvhq,5,1)

ans =

vvuttcccqgcunjtpjgkuqpknjkygkkgcjfqrkqjrqudukv
pfncwuce
```

We now do a frequency count of the preceding substring. To do this, we use the *frequency* function and use ans as input. In MATLAB, if a command is issued without declaring a variable for

the result, MATLAB will put the output in the variable ans.

```
>> frequency(ans);


 a   0
 b   0
 c   7
 d   1
 e   1
 f   2
 g   9
 h   0
 i   1
 j   8
 k   8
 l   0
 m   0
 n   3
 o   0
 p   4
 q   5
 r   2
 s   0
 t   3
 u   6
```

```
v   5
w   1
x   0
y   1
z   0
```

To express this as a vector of frequencies, we use the *vigvec* function. The *vigvec* function will not only display the frequency counts just shown, but will return a vector that contains the frequencies. In the following output, we have suppressed the table of frequency counts since they appear above and have reported the results in the *short* format.

```
>> vigvec(vvhq,5,1)

ans =
   0
   0
   0.1045
   0.0149
   0.0149
```

```
0.0299
0.1343
0
0.0149
0.1194
0.1194
0
0
0.0448
0
0.0597
0.0746
0.0299
0
0.0448
0.0896
0.0746
0.0149
0
0.0149
0
```

(If we are working in rational format, these numbers are displayed as rationals.) The dot products of this vector with the displacements of the

alphabet frequency vector are computed as follows:

```
>> corr(ans)

ans =
   0.0250
   0.0391
   0.0713
   0.0388
   0.0275
   0.0380
   0.0512
   0.0301
   0.0325
   0.0430
   0.0338
   0.0299
   0.0343
   0.0446
   0.0356
   0.0402
   0.0434
   0.0502
   0.0392
   0.0296
   0.0326
```

```
    0.0392
    0.0366
    0.0316
    0.0488
    0.0349
```

The third entry is the maximum, but sometimes the largest entry is hard to locate. One way to find it is

```
>> max(ans)

ans =
    0.0713
```

Now it is easy to look through the list and find this number (it usually occurs only once). Since it occurs in the third position, the first shift for this Vigenère cipher is by 2, corresponding to the letter *c*. A procedure similar to the one just used (using *vigvec(vvhq, 5,2), . . . ,*

vigvec(vvhq,5,5)) shows that the other shifts are probably 14, 3, 4, 18. Let's check that we have the correct key by decrypting.

```
>> vigenere(vvhq,-[2,14,3,4,18])

ans =

themethodusedforthepreparationandreadingofcode
extremeandatthesametimeimpossibleoftranslation
eeasewithwhichthekeymaybechangedisanotherpoint
fthiscodebythosedesiringtotransmitimportantmes
estdangeroftheirmessagesbeingreadbypoliticalor
```

For the record, the plaintext was originally encrypted by the command

```
>> vigenere(ans,[2,14,3,4,18])

ans =

vvhqwvvrhmusgjgthkihtssejchlsfcbgvwcrlryqtfsvg
hbltspisprdxljsveeghlqwkasskuwepwqtwvspgoelkcq
```

```
wgoviokhkazkqkxzgyhcecmeiujoqkwfwvefqhkijrclrl
twlauqrhwdmwlgusgikkflryvcwvspgpmlkassjvoqxegg
ikvrdrygfrjljslveggveyggeiapuuisfpbtgnwwmuczrv
```

C.3 Examples for Chapter 3

Example 5

Find gcd(23456; 987654).

```
>> gcd(23456,987654)

ans =
   2
```

If larger integers are used, they should be expressed in symbolic mode; otherwise, only the first 16 digits of the entries are used accurately. The present calculation could have been done as

```
>> gcd(sym('23456'),sym('987654'))
```

```
ans =
   2
```

Example 6

Solve 23456x+987654y=2 in integers x, y.

```
>> [a,b,c]=gcd(23456,987654)

a =
   2

b =
   -3158

c =
   75
```

This means that 2 is the gcd and $23456 \cdot (-3158) + 987654 \cdot 75 = 2$.

Example 7

Compute 234·456 (mod 789).

```
>> mod(234*456,789)

ans =
   189
```

Example 8

Compute 234567876543 (mod 565656565).

```
>>
powermod(sym('234567'),sym('876543'),sym('5656

ans =
   5334
```

Example 9

Find the multiplicative inverse of

87878787 (mod 9191919191).

```
>>
invmodn(sym('87878787'),sym('9191919191'))

ans =
   7079995354
```

Example 10

Solve 7654x ≡ 2389 (mod 65537).

SOLUTION

To solve this problem, we follow the method described in Section 3.3. We calculate 7654−1 and then multiply it by 2389:

```
>> invmodn(7654,65537)
```

```
ans =
   54637

>> mod(ans*2389,65537)

ans =
   43626
```

Example 11

Find x with

$x \equiv 2 \pmod{78}$, $x \equiv 5 \pmod{97}$, $x \equiv 1 \pmod{119}$.

SOLUTION

To solve the problem we use the function *crt*.

```
>> crt([2 5 1],[78 97 119])

ans =
   647480
```

We can check the answer:

```
>> mod(647480,[78 97 119])

ans =
    2    5    1
```

Example 12

Factor 123450 into primes.

```
>> factor(123450)

ans =
    2    3    5    5    823
```

This means that 123450=2131528231.

Example 13

Evaluate $\phi(12345)$.

```
>> eulerphi(12345)

ans =
   6576
```

Example 14

Find a primitive root for the prime 65537.

```
>> primitiveroot(65537)

ans =
   3
```

Therefore, 3 is a primitive root for 65537.

Example 15

Find the inverse of the matrix

131235415362716810 (mod999).

SOLUTION

First, we enter the matrix as M.

```
>> M=[13 12 35; 41 53 62; 71 68
10];
```

Next, invert the matrix without the mod:

```
>> Minv=inv(M)

Minv =
     233/2158     -539/8142
103/3165
     -270/2309    139/2015
-40/2171
     209/7318     32/34139
-197/34139
```

We need to multiply by the determinant of M in order to clear the fractions out

of the numbers in Minv. Then we need to multiply by the inverse of the determinant mod 999.

```
>> Mdet=det(M)

Mdet =
   -34139

>> invmodn(Mdet,999)

ans =
   589
```

The answer is given by

```
>> mod(Minv*589*Mdet,999)

ans =
    772    472    965
    641    516    851
    150    133    149
```

Therefore, the inverse matrix mod 999

is 772472965641516851150133149.

In many cases, it is possible to determine by inspection the common denominator that must be removed. When this is not the case, note that the determinant of the original matrix will always work as a common denominator.

Example 16

Find a square root of 26951623672 mod the prime p=98573007539.

SOLUTION

Since $p \equiv 3 \pmod 4$, we can use the proposition of Section 3.9:

```
>> powermod(sym('26951623672'),
(sym('98573007539')+1)/4,sym('98573007539'))
```

```
ans =
   98338017685
```

The other square root is minus this one:

```
>> mod(-ans,32579)

ans =
   234989854
```

Example 17

Let n=34222273=9803·3491. Find all four solutions of x2 ≡ 19101358 (mod 34222273).

SOLUTION

First, find a square root mod each of the two prime factors, both of which are

congruent to 3 (mod 4):

```
>> powermod(19101358,
(9803+1)/4,9803)

ans =
   3998
>> powermod(19101358,
(3491+1)/4,3491)

ans =
   1318
```

Therefore, the square roots are congruent to ±3998 (mod 9803) and are congruent to ±1318 (mod 3491). There are four ways to combine these using the Chinese remainder theorem:

```
>> crt([3998 1318],[9803 3491])

ans =
   43210
```

```
>> crt([-3998 1318],[9803 3491])

ans =
   8397173

>> crt([3998 -1318],[9803 3491])

ans =
   25825100

>> crt([-3998 -1318],[9803 3491])

ans =
   34179063
```

These are the four desired square roots.

C.4 Examples for Chapter 5

Example 18

Compute the first 50 terms of the recurrence

$$x_{n+5} \equiv x_n + x_{n+2} \pmod 2.$$

The initial values are 0, 1, 0, 0, 0.

SOLUTION

The vector of coefficients is [1, 0, 1, 0, 0] and the initial values are given by the vector [0, 1, 0, 0, 0]. Type

```
>> lfsr([1 0 1 0 0],[0 1 0 0 0],50)

ans =
```

```
    Columns 1 through 12
     0   1   0   0   0   0   1   0
0   1   0   1
    Columns 13 through 24
     1   0   0   1   1   1   1   1
0   0   0   1
    Columns 25 through 36
     1   0   1   1   1   0   1   0
1   0   0   0
    Columns 37 through 48
     0   1   0   0   1   0   1   1
0   0   1   1
    Columns 49 through 50
     1   1
```

Example 19

Suppose the first 20 terms of an LFSR sequence are 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1. Find a recursion that generates this sequence.

SOLUTION

First, we find a candidate for the length of the recurrence. The command *lfsrlength(v, n)* calculates the determinants mod 2 of the first n matrices that appear in the procedure described in Section 5.2 for the sequence v. Recall that the last nonzero determinant gives the length of the recurrence.

```
>> lfsrlength([1 0 1 0 1 1 1 0 0 0
0 1 1 1 0 1 0 1 0 1],10)
    Order Determinant
       1         1
       2         1
       3         0
       4         1
       5         0
       6         1
       7         0
       8         0
       9         0
      10         0
```

The last nonzero determinant is the sixth one, so we guess that the recurrence has length 6. To find the coefficients:

```
>> lfsrsolve([1 0 1 0 1 1 1 0 0 0 0
1 1 1 0 1 0 1 0 1],6)

ans =
    1   0   1   1   1   0
```

This gives the recurrence as

$$x_{n+6} \equiv x_n + x_{n+2} + x_{n+3} + x_{n+4} \pmod 2.$$

Example 20

The ciphertext 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0 was produced by adding the output of a LFSR onto the plaintext mod 2 (i.e., XOR the plaintext with the LFSR

output). Suppose you know that the plaintext starts 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0. Find the rest of the plaintext.

SOLUTION

XOR the ciphertext with the known part of the plaintext to obtain the beginning of the LFSR output:

```
>> x=mod([1 1 1 1 1 1 0 0 0 0 0 0 1
1 1 0 0]+[0 1 1 0 1 0 1 0 1 0 0 1 1
0 0 0 1],2)

x =
   Columns 1 through 12
    1   0   0   1   0   1   1   0
1   0   0   1
   Columns 13 through 17
    0   1   1   0   1
```

This is the beginning of the LFSR

output. Let's find the length of the recurrence:

```
>> lfsrlength(x,8)
    Order  Determinant
      1          1
      2          0
      3          1
      4          0
      5          1
      6          0
      7          0
      8          0
```

We guess the length is 5. To find the coefficients of the recurrence:

```
>> lfsrsolve(x,5)

ans =
    1   1   0   0   1
```

Now we can generate the full output of the LFSR using the coefficients we just

found plus the first five terms of the LFSR output:

```
>> lfsr([1 1 0 0 1],[1 0 0 1 0],40)

ans =
   Columns 1 through 12
    1   0   0   1   0   1   1   0
1   0   0   1
   Columns 13 through 24
    0   1   1   0   1   0   0   1
0   1   1   0
   Columns 25 through 36
    1   0   0   1   0   1   1   0
1   0   0   1
   Columns 37 through 40
    0   1   1   0
```

When we XOR the LFSR output with the ciphertext, we get back the plaintext:

```
>> mod(ans+[0 1 1 0 1 0 1 0 1 0 0 1
1 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 0 0 1 0 1 1 0],2)
```

```
ans =
   Columns 1 through 12
    1   1   1   1   1   1   0   0
0   0   0   0
   Columns 13 through 24
    1   1   1   0   0   0   1   1
1   1   0   0
   Columns 25 through 36
    0   0   1   1   1   1   1   1
1   0   0   0
   Columns 37 through 40
    0   0   0   0
```

This is the plaintext.

C.5 Examples for Chapter 6

Example 21

The ciphertext

22, 09, 00, 12, 03, 01, 10, 03, 04, 08, 01, 17

was encrypted using a Hill cipher with matrix

1234567810.

Decrypt it.

SOLUTION

A matrix abcd is entered as [a, b;c, d]. Type M*N to multiply matrices M and N. Type v*M to multiply a vector v on

the right by a matrix M.

First, we put the above matrix in the variable M.

```
>> M=[1 2 3; 4 5 6; 7 8 10]

M =
```

```
1234567810
```

Next, we need to invert the matrix mod 26:

```
>> Minv=inv(M)

   Minv =
```

```
-2/3-4/31-2/311/3-21-21
```

Since we are working mod 26, we can't stop with numbers like 2/3. We need to get rid of the denominators and reduce mod 26. To do so, we multiply by 3 to extract the numerators of the fractions, then multiply by the inverse of 3 mod 26 to put the "denominators" back in (see Section 3.3):

```
>> M1=Minv*3

M1 =
```

```
-2-43-211-63-63
```

```
>> M2=mod(round(M1*9,26))

M2 =
```

```
8161821241241
```

Note that we used the function *round* in calculating M2. This was done since MATLAB performs its calculations in floating point and calculating the inverse matrix Minv produces numbers that are slightly different from whole numbers. For example, consider the following:

```
>> a=1.99999999;display([a,
mod(a,2), mod(round(a),2)])
```

```
2.0000 2.0000 0
```

The matrix M2 is the inverse of the matrix M mod 26. We can check this as follows:

```
>> mod(M2*M,26)
ans =
    1   0   0
    0   1   0
```

```
     0    0    1
```

To decrypt, we break the ciphertext into blocks of three numbers and multiply each block on the right by the inverse matrix we just calculated:

```
>> mod([22,9,0]*M2,26)

ans =
    14    21    4

>> mod([12,3,1]*M2,26)

ans =
    17    19    7

>> mod([10,3,4]*M2,26)

ans =
    4    7    8

>> mod([8,1,17]*M2,26)

ans =
    11    11    23
```

Therefore, the plaintext is 14, 21, 4, 17, 19, 7, 4, 7, 8, 11, 11, 23. This can be changed back to letters:

```
>> int2text([14 21 4 17 19 7 4 7 8
11 11 23])

ans =

overthehillx
```

Note that the final `x` was appended to the plaintext in order to complete a block of three letters.

C.6 Examples for Chapter 9

Example 22

Two functions, *nextprime* and *randprime*, can be used to generate prime numbers. The function *nextprime* takes a number n as input and attempts to find the next prime after n. The function *randprime* takes a number n as input and attempts to find a random prime between 1 and n. It uses the Miller-Rabin test described in Chapter 9.

```
>> nextprime(346735)

ans =
   346739
```

```
>> randprime(888888)

ans =
   737309
```

For larger inputs, use symbolic mode:

```
>> nextprime(10^sym(60))

ans =

1000000000000000000000000000000000000000000000000

>> randprime(10^sym(50))

ans =

58232516535825662451486550708068534731864929191
```

It is interesting to note the difference that the `,` makes when entering a large integer:

```
>>
```

```
nextprime(sym('12345678901234567890123456789'

ans =
   12345678901234567890123456907

>>
nextprime(sym(12345678901234567890123456789))

ans =
   12345678901234567877719597071
```

In the second case, the input was a number, so only the first 16 digits of the input were used correctly when changing to symbolic mode, while the first case regarded the entire input as a string and therefore used all of the digits.

Example 23

Suppose you want to change the text `hellohowareyou` to numbers:

```
>> text2int1('hellohowareyou')

ans =
   805121215081523011805251521
```

Note that we are now using $a = 1$, $b = 2$, ..., $z = 26$, since otherwise a's at the beginnings of messages would disappear. (A more efficient procedure would be to work in base 27, so the numerical form of the message would be $8+5\cdot27+12\cdot272+\cdots+21\cdot2713=87495221502384554951$. Note that this uses fewer digits.)

Now suppose you want to change it back to letters:

```
>>
int2text1(805121215081523011805251521)

ans =
```

```
    'hellohowareyou'
```

Example 24

Encrypt the message `hi` using RSA with n=823091 and e=17.

SOLUTION

First, change the message to numbers:

```
>> text2int1('hi')

ans =
   809
```

Now, raise it to the eth power mod n:

```
>> powermod(ans,17,823091)

ans =
   596912
```

Example 25

Decrypt the ciphertext in the previous problem.

SOLUTION

First, we need to find the decryption exponent d. To do this, we need to find $\phi(823091)$. One way is

```
>> eulerphi(823091)

ans =
   821184
```

Another way is to factor n as p·q and then compute (p−1)(q−1):

```
>> factor(823091)
```

```
ans =
    659   1249

>> 658*1248

ans =
   821184
```

Since $de \equiv 1 \pmod{\phi(n)}$, we compute the following (note that we are finding the inverse of e mod $\phi(n)$, not mod n):

```
>> invmodn(17,821184)

ans =
   48305
```

Therefore, d=48305. To decrypt, raise the ciphertext to the dth power mod n:

```
>> powermod(596912,48305,823091)

ans =
   809
```

Finally, change back to letters:

```
>> int2text1(ans)

ans =
   hi
```

Example 26

Encrypt `hellohowareyou` using RSA with n=823091 and e=17.

SOLUTION

First, change the plaintext to numbers:

```
>> text2int1('hellohowareyou')

ans =
   805121215081523011805251521
```

Suppose we simply raised this to the eth

power mod n:

```
>> powermod(ans,17,823091)

ans =
   447613
```

If we decrypt (we know d from Example 25), we obtain

```
>> powermod(ans,48305,823091)

ans =
   628883
```

This is not the original plaintext. The reason is that the plaintext is larger than n, so we have obtained the plaintext mod n:

```
>>
mod(text2int1('hellohowareyou'),823091)
```

```
ans =
   628883
```

We need to break the plaintext into blocks, each less than n. In our case, we use three letters at a time:

805121215081523011805251521

```
>> powermod(80512,17,823091)

ans =
   757396

>> powermod(121508,17,823091)

ans =
   164513

>> powermod(152301,17,823091)

ans =
   121217

>> powermod(180525,17,823091)

ans =
   594220
```

```
>> powermod(1521,17,823091)

ans =
   442163
```

The ciphertext is therefore 757396164513121217594220442163. Note that there is no reason to change this back to letters. In fact, it doesn't correspond to any text with letters.

Decrypt each block individually:

```
>> powermod(757396,48305,823091)

ans =
   80512

>> powermod(164513,48305,823091)

ans =
   121508
```

Etc.

We'll now do some examples with large numbers, namely the numbers in the RSA Challenge discussed in Section 9.5. These are stored under the names *rsan, rsae, rsap, rsaq*:

```
>> rsan

ans =

114381625757888867669235779976146612010218296
706935245733897830597123563958705058989075147

>> rsae

ans =
   9007
```

Example 27

Encrypt each of the messages *b, ba, bar, bard* using *rsan* and *rsae*.

```
>> powermod(text2int1('b'), rsae,
rsan)

ans =

709467584676126685983701649915507861828763310
867822617164972001221553323484620140532879875

>> powermod(text2int1('ba'), rsae,
rsan)

ans =

350451306089751003250117094498719542737882047
276217598060279622705380315655647733520336717

>> powermod(txt2int1('bar'), rsae,
rsan)

ans =

448145128638551010760045308594921093424295316
008436459868804059531028183128225863625802987
```

```
>> powermod(text2int1('bard'),
rsae, rsan)

ans =

24238077785111666423202862512090317393485212955
56054323297179804928958073445752663026449873980
```

Observe that the ciphertexts are all the same length. There seems to be no easy way to determine the length of the corresponding plaintext.

Example 28

Using the factorization rsan=rsap·rsaq, find the decryption exponent for the RSA Challenge, and decrypt the ciphertext (see Section 9.5).

SOLUTION

First, we find the decryption exponent:

```
>> rsad=invmodn(rsae,-1,
(rsap-1)*(rsaq-1));
```

Note that we use the final semicolon to avoid printing out the value. If you want to see the value of *rsad*, see Section 9.5, or don't use the semicolon. To decrypt the ciphertext, which is stored as *rsaci*, and change to letters:

```
>> int2text1(powermod(rsaci, rsad,
rsan))

ans =
   the magic words are squeamish
ossifrage
```

Example 29

Encrypt the message *rsaencryptsmessageswell* using *rsan* and *rsae*.

```
>> ci =
powermod(text2int1('rsaencryptsmessageswell'),
rsae, rsan)
   ci =

9463942034900225931630582353924949641464096993
7195065425436558490601396632881775353928311265
```

We called the ciphertext `ci` because we need it in Example 30.

Example 30

Decrypt the preceding ciphertext.

SOLUTION

Fortunately, we know the decryption

exponent *rsad*. Therefore, we compute

```
>> powermod(ans, rsad, rsan)

ans =

1819010514031825162019130519190107051923051212

>> int2text1(ans)

ans =
   rsaencryptsmessageswell
```

Suppose we lose the final 4 of the ciphertext in transmission. Let's try to decrypt what's left (subtracting 4 and dividing by 10 is a mathematical way to remove the 4): ‘ `>> powermod((ci - 4)/10, rsad, rsan)`

```
ans =

47952999173195988664902352629525486409113633 89
```

```
05884123003734879696577942541171589569212679125
```

If we try to change this to letters, we get a weird-looking answer. A small error in the plaintext completely changes the decrypted message and usually produces garbage.

Example 31

Suppose we are told that n=113137712755903125 67 is the product of two primes and that ϕ(n)=113137711876087444 00. Factor n.

SOLUTION

We know (see Section 9.1) that p and q are the roots of X2−(n−ϕ(n)+1)X+n.

Therefore, we compute (`vpa` is for variable precision arithmetic)

```
>> digits(50); syms y;
vpasolve(y^2-
(sym('113137712755903125 67') -
sym('11313771187608744400')+1)*y
+sym('113137712755903125 67'),y)

ans =
   128781017.0 87852787151.0
```

Therefore, n=128781017·87852787151. We also could have used the quadratic formula to find the roots.

Example 32

Suppose we know *rsae* and *rsad*. Use these to factor *rsan*.

SOLUTION

We use the ar ≡ 1 factorization method from Section 9.4. First write rsae·rsad −1=2sm with m odd. One way to do this is first to compute

```
>> rsae*rsad - 1

ans =

961034419617782266156919023359583834109854129O
411559855750873526591561748985573429951315946B

>> ans/2

ans =

480517209808891133078459511679791917054927064S
205779927875436763295780874492786714975657973A

>> ans/2

ans =

240258604904445566539229755839895958527463532Z
102889963937718381647890437246393357487828986Z
```

We continue this way for six more steps until we get

```
ans =

375404070163196197717546493499837435199161769153576556865268497132482880819748962107473279175
```

This number is m. Now choose a random integer a. Hoping to be lucky, we choose 13. As in the ar ≡ 1 factorization method, we compute

```
>>b0=powermod(13, ans, rsan)

b0 =

275743685070065305922434948688471611984230957070301831098398623708005293380929847954901926431
```

Since this is not ±1 (mod rsan), we successively square it until we get ±1:

```
>> b1=powermod(b0,2,rsan)

b1 =

48318960321928515580138476418723034554104099066
66549964125829556360352661561086864311942985744

>> b2=powermod(b1,2,rsan)

b2 =

78172814154877356579141928058754000021948787054
52151818397420560132755219134875609447320735164

>> b3=powermod(b2, 2, rsan)

b3 =

42836191202508728742199299040582900202976222914
65094445182394621863794705694420551013929922934

>> b4=powermod(b3, 2, rsan)

b4 =
   1
```

Since the last number before the 1 was not $\pm 1 \pmod{\text{rsan}}$, we have an example

of $x \equiv \pm 1 \pmod{rsan}$ with $x2 \equiv 1$. Therefore, $\gcd(x-1, rsan)$ is a nontrivial factor of *rsan*:

```
>> gcd(b3 - 1, rsan)

ans =

32769132993266709549961988190834461413177642967992942539798288533
```

This is *rsaq*. The other factor is obtained by computing *rsan/rsaq*:

```
>> rsan/ans

ans =

34905295108476509491478496199038981334177646382922413636727137066
```

This is *rsap*.

Example 33

Suppose you know that

$1508834755694512 \equiv 168875705328582 \pmod{205611444308117}$.

Factor 205611444308117.

SOLUTION

We use the Basic Principle of Section 9.4.

```
>> g=
gcd(150883475569451-16887570532858,2056114443
0

g =
   23495881
```

This gives one factor. The other is

```
>> 205611444308117/g

ans =
   8750957
```

We can check that these factors are actually primes, so we can't factor any further:

```
>> primetest(ans)

ans =
   1

>> primetest(g)

ans =
   1
```

Example 34

Factor n=376875575426394855599989992897873239 by the p−1 method.

SOLUTION

Let’s choose our bound as B=100, and let’s take a=2, so we compute $2^{100!} \pmod{n}$:

```
>>
powermod(2,factorial(100),sym('376875754263942
2897873239')

ans =

369676678301956331939422106251199512
```

Then we compute the gcd of $2^{100!}-1$ and n:

```
>> gcd(ans - 1,
sym'(37687557542639485559998999289787323 9')

ans =
   430553161739796481
```

This is a factor p. The other factor q is

```
>>
sym('37687557542639485559998999289787323 9')/
ans

ans =
   875328783798732119
```

Let's see why this worked. The factorizations of p−1 and q−1 are

```
>> factor(sym('4305531617397964 81')
- 1)

ans =
   [ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3,
3, 3, 3, 5, 7, 7, 7, 7, 11, 11, 11,
47]

>> factor(sym('8753287837987321 19')
- 1)

ans =
   [ 2, 61, 20357, 39301, 8967967]
```

We see that 100! is a multiple of p−1, so

$2100! \equiv 1 \pmod{p}$. However, 100! is not a multiple of $q-1$, so it is likely that $2100! \equiv 1 \pmod{q}$. Therefore, both $2100!-1$ and pq have p as a factor, but only pq has q as a factor. It follows that the gcd is p.

C.7 Examples for Chapter 10

Example 35

Let’s solve the discrete log problem $2x \equiv 71 \pmod{131}$ by the Baby Step-Giant Step method of Subsection 10.2.2. We take N=12 since N2>p−1=130 and we form two lists. The first is 2j (mod 1)31 for 0 ≤ j ≤ 11:

```
>> for k=0:11;z=[k,
powermod(2,k,131)];disp(z);end;

0  1
1  2
2  4
3  8
4  16
```

```
5 32
6 64
7 128
8 125
9 119
10 107
11 83
```

The second is $71 \cdot 2^{-j} \pmod{131}$ for $0 \le j \le 11$:

```
>> for k=0:11;z=[k,
mod(71*invmodn(powermod(2,12*k,131),131),131)]
disp(z);end;

0 71
1 17
2 124
3 26
4 128
5 86
6 111
7 93
8 85
9 96
10 130
11 116
```

The number 128 is on both lists, so we see that $27 \equiv 71 \cdot 2 - 12 * 4 \pmod{131}$. Therefore,

$$71 \equiv 27 + 4 * 12 \equiv 255 \pmod{131}.$$

C.8 Examples for Chapter 12

Example 36

Suppose there are 23 people in a room. What is the probability that at least two have the same birthday?

SOLUTION

The probability that no two have the same birthday is $\prod i=122(1–i/365)$ (note that the product stops at i=22, not i=23). Subtracting from 1 gives the probability that at least two have the same birthday:

```
>> 1-prod( 1 - (1:22)/365)
```

```
ans =
   0.5073
```

Example 37

Suppose a lazy phone company employee assigns telephone numbers by choosing random seven-digit numbers. In a town with 10,000 phones, what is the probability that two people receive the same number?

```
>> 1-prod( 1 - (1:9999)/10^7)

ans =
   0.9933
```

Note that the number of phones is about three times the square root of the number of possibilities. This means that

we expect the probability to be high, which it is. From Section 12.1, we have the estimate that if there are around $2(\ln 2)107 \approx 3723$ phones, there should be a 50% chance of a match. Let's see how accurate this is:

```
>> 1-prod( 1 - (1:3722)/10^7)

ans =
   0.4999
```

C.9 Examples for Chapter 17

Example 38

Suppose we have a (5, 8) Shamir secret sharing scheme. Everything is mod the prime p=987541. Five of the shares are

(9853, 853), (4421, 4387), (6543, 1234), (93293, 78428), (12398, 7563).

Find the secret.

SOLUTION

The function *interppoly(x,f,m)* calculates the interpolating polynomial that passes through the points (xj, fj). The arithmetic is done mod m.

In order to use this function, we need to make a vector that contains the x values, and another vector that contains the share values. This can be done using the following two commands:

```
>> x=[9853 4421 6543 93293 12398];

>> s=[853 4387 1234 78428 7563];
```

Now we calculate the coefficients for the interpolating polynomial.

```
>> y=interppoly(x,s,987541)

y =
    678987   14728   1651   574413
 456741
```

The first value corresponds to the constant term in the interpolating polynomial and is the secret value.

Therefore, 678987 is the secret.

C.10 Examples for Chapter 18

Example 39

Here is a game you can play. It is essentially the simplified version of poker over the telephone from Section 18.2. There are five cards: `ten`, `jack`, `queen`, `king`, `ace`. We have chosen to abbreviate them by the following: ten, ace, que, jac, kin. They are shuffled and disguised by raising their numbers to a random exponent mod the prime 300649. You are supposed to guess which one is the ace.

First, the cards are entered in and

converted to numerical values by the following steps:

```
>>
cards=['ten';'ace';'que';'jac';'kin'];

>> cvals=text2int1(cards)

cvals =

   200514
   10305
   172105
   100103
   110914
```

Next, we pick a random exponent k that will be used in the hiding operation. We use the semicolon after *khide* so that we cannot cheat and see what value of k is being used.

```
>> p=300649;
```

```
>> k=khide(p);
```

Now, shuffle the disguised cards (their numbers are raised to the kth power mod p and then randomly permuted):

```
>> shufvals=shuffle(cvals,k,p)

shufvals =
    226536
    226058
    241033
    281258
    116809
```

These are the five cards. None looks like the ace; that's because their numbers have been raised to powers mod the prime. Make a guess anyway. Let's see if you're correct.

```
>> reveal(shufvals,k,p)
```

```
ans =


jac
que
ten
kin
ace
```

Let’s play again:

```
>> k=khide(p);

» shufvals=shuffle(cvals,k,p)

shufvals =
    117135
    144487
    108150
    266322
    264045
```

Make your guess (note that the numbers are different because a different random exponent was used). Were you lucky?

```
>> reveal(shufvals,k,p)

ans =

kin
jac
ten
que
ace
```

Perhaps you need some help. Let's play one more time:

```
>> k=khide(p);

» shufvals=shuffle(cvals,k,p)

shufvals =
    108150
    144487
    266322
    264045
    117135
```

We now ask for advice:

```
>> advise(shufvals,p);
```

```
Ace Index: 4
```

We are advised that the fourth card is the ace. Let's see:

```
>> reveal(shufvals,k,p)

ans =


ten
jac
que
ace
kin
```

How does this work? Read the part on "How to Cheat" in Section 18.2. Note that if we raise the numbers for the cards to the (p−1)/2 power mod p, we get

```
```

```
>> powermod(cvals,(p-1)/2,p)

ans =
      1
   300648
      1
      1
      1
```

Therefore, only the ace is a quadratic nonresidue mod p.

C.11 Examples for Chapter 21

Example 40

We want to graph the elliptic curve $y2=x(x-1)(x+1)$.

First, we create a string v that contains the equation we wish to graph.

```
>> v='y^2 - x*(x-1)*(x+1)';
```

Next we use the *ezplot* command to plot the elliptic curve.

```
>> ezplot(v,[-1,3,-5,5])
```

The plot appears in Figure C.1. The use of [−1, 3, −5, 5] in the preceding command is to define the limits of the x-axis and y-axis in the plot.

Figure C.1 Graph of the Elliptic Curve y2=x(x−1)(x+1).

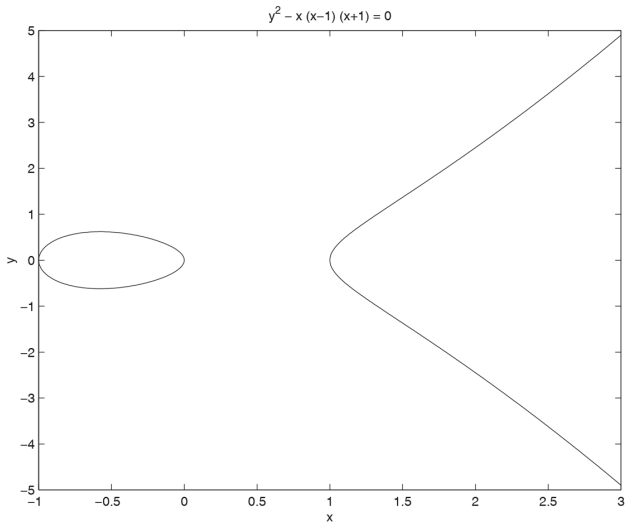


Figure C.1 Full Alternative Text

Example 41

Add the points (1,3) and (3,5) on the

elliptic curve $y2 \equiv x3+24x+13 \pmod{29}$.

```
>> addell([1,3],[3,5],24,13,29)

ans =
    26     1
```

You can check that the point (26,1) is on the curve: $263+24 \cdot 26+13 \equiv 12 \pmod{29}$. (*Note:* `addell([x,y],[u,v],b,c,n)` is only programmed to work for odd n.)

Example 42

Add (1,3) to the point at infinity on the curve of the previous example.

```
>> addell([1,3],[inf,inf],24,13,29)

ans =
    1     3
```

As expected, adding the point at infinity to a point P returns the point P.

Example 43

Let P=(1, 3) be a point on the elliptic curve y2 ≡ x3+24x+13 (mod 29). Find 7P.

```
>> multell([1,3],7,24,13,29)

ans =
    15     6
```

Example 44

Find k(1, 3) for k=1, 2, 3, …, 40 on the curve of the previous example.

```
>> multsell([1,3],40,24,13,29)
```

```
ans =
    1:     1      3
    2:     11     10
    3:     23     28
    4:     0      10
    5:     19     9
    6:     18     19
    7:     15     6
    8:     20     24
    9:     4      12
   10:     4      17
   11:     20     5
   12:     15     23
   13:     18     10
   14:     19     22
   15:     0      19
   16:     23     1
   17:     11     19
   18:     1      26
   19:    inf    Inf
   20:     1      3
   21:     10     10
   22:     23     28
   23:     0      10
   24:     19     7
   25:     18     19
   26:     15     6
   27:     20     24
   28:     4      12
```

```
29:    4     17
30:    20    5
31:    15    23
32:    18    10
33:    19    22
34:    0     19
35:    23    1
36:    11    19
37:    1     26
38:   inf   inf
39:    1     3
40:    10    10
```

Notice how the points repeat after every 19 multiples.

Example 45

The previous four examples worked mod the prime 29. If we work mod a composite number, the situation at infinity becomes more complicated since we could be at infinity mod both factors or we could be at infinity mod

one of the factors but not mod the other. Therefore, we stop the calculation if this last situation happens and we exhibit a factor. For example, let's try to compute 12P, where P=(1, 3) is on the elliptic curve $y2 \equiv x3-5x+13 \pmod{209}$:

```
>> multell([1,3],12,-5,13,11*19)

Elliptic Curve addition produced a
factor of n, factor= 19
Multell found a factor of n and
exited

ans =
   []
```

Now let's compute the successive multiples to see what happened along the way:

```
>> multsell([1,3],12,-5,13,11*19)
```

```
Elliptic Curve addition produced a
factor of n, factor= 19
Multsell ended early since it found
a factor

ans =
1:   1     3
2:   91    27
3:   118   133
4:   148   182
5:   20    35
```

When we computed 6P, we ended up at infinity mod 19. Let's see what is happening mod the two prime factors of 209, namely 19 and 11:

```
>> multsell([1,3],20,-5,13,19)

ans =
 1:    1     3
 2:   15     8
 3:    4     0
 4:   15    11
 5:    1    16
 6:   Inf   Inf
```

```
 7:    1     3
 8:   15     8
 9:    4     0
10:   15    11
11:   15     8
12:   Inf   Inf
13:    1     3
14:   15     8
15:    4     0
16:   15    11
17:    1    16
18:   Inf   Inf
19:    1     3
20:   15     8
```

```
>> multsell([1,3],20,-5,13,11)

ans =
 1:    1     3
 2:    3     5
 3:    8     1
 4:    5     6
 5:    9     2
 6:    6    10
 7:    2     0
 8:    6     1
 9:    9     9
10:    5     5
```

```
11:    8    10
12:    3     6
13:    1     8
14:  Inf   Inf
15:    1     3
16:    3     5
17:    8     1
18:    5     6
19:    9     2
20:    6    10
```

After six steps, we were at infinity mod 19, but it takes 14 steps to reach infinity mod 11. To find 6P, we needed to invert a number that was 0 mod 19 and nonzero mod 11. This couldn't be done, but it yielded the factor 19. This is the basis of the elliptic curve factorization method.

Example 46

Factor 193279 using elliptic curves.

SOLUTION

First, we need to choose some random elliptic curves and a point on each curve. For example, let's take P=(2, 4) and the elliptic curve

$$y2 \equiv x3-10x+b \pmod{193279}.$$

For P to lie on the curve, we take b=28. We'll also take

$$y2 \equiv x3+11x-11, P=(1, 1), y2 \equiv x3+17x-14, P=(1, 2).$$

Now we compute multiples of the point P. We do the analog of the p−1 method, so we choose a bound B, say B=12, and compute B!P.

```
>>
multell([2,4],factorial(12),-10,28,193279)

Elliptic Curve addition produced a
```

```
factor of n, factor= 347
Multell found a factor of n and
exited

ans =
   []

>>
multell([1,1],factorial(12),11,-11,193279)

ans =
    13862    35249

»
multell([1,2],factorial(12),17,-14,193279)

Elliptic Curve addition produced a
factor of n, factor= 557
Multell found a factor of n and
exited

ans =
   []
```

Let’s analyze in more detail what happened in these examples.

On the first curve, 266P ends up at infinity mod 557 and 35P is infinity mod 347. Since 272=2·7·9, it has a prime factor larger than B=12, so B!P is not infinity mod 557. But 35 divides B!, so B!P is infinity mod 347.

On the second curve, 356P=infinity mod 347, and 561P=infinity mod 557. Since 356=4·89 and 561=3·11·17, we don't expect to find the factorization with this curve.

The third curve is a surprise. We have 331P=infinity mod 347 and 272P=infinity mod 557. Since 331 is prime and 272=16·17, we don't expect to find the factorization with this curve. However, by chance, an intermediate step in the calculation of B!P yielded the

factorization. Here's what happened. At an intermediate step in the calculation, the program required adding the points (184993, 13462) and (20678, 150484). These two points are congruent mod 557 but not mod 347. Therefore, the slope of the line through these two points is defined mod 347 but is 0/0 mod 557. When we tried to find the multiplicative inverse of the denominator mod 193279, the gcd algorithm yielded the factor 557. This phenomenon is fairly rare.

Example 47

Here is how to produce the example of an elliptic curve ElGamal cryptosystem from Section 21.5. For more details, see the text. The elliptic curve is $y2 \equiv x3+3x+45 \pmod{8831}$ and the point is

G=(4, 11). Alice's message is the point Pm=(5, 1743).

Bob has chosen his secret random number aB=3 and has computed aBG:

```
>> multell([4,11],3,3,45,8831)

ans =
    413    1808
```

Bob publishes this point. Alice chooses the random number k=8 and computes kG and Pm+k(aBG):

```
>> multell([4,11],8,3,45,8831)

ans =
    5415    6321

>>
addell([5,1743],multell([413,1808],8,3,45,8831

ans =
```

```
     6626    3576
```

Alice sends (5415,6321) and (6626, 3576) to Bob, who multiplies the first of these point by aB:

```
>> multell([5415,6321],3,3,45,8831) ans
= 673 146
```

Bob then subtracts the result from the last point Alice sends him. Note that he subtracts by adding the point with the second coordinate negated:

```
>> addell([6626,3576],
[673,-146],3,45,8831)

ans =
   5 1743
```

Bob has therefore received Alice's message.

Example 48

Let’s reproduce the numbers in the example of a Diffie-Hellman key exchange from Section 21.5: The elliptic curve is $y2 \equiv x3+x+7206 \pmod{7211}$ and the point is G=(3, 5). Alice chooses her secret NA=12 and Bob chooses his secret NB=23. Alice calculates

```
>> multell([3,5],12,1,7206,7211)

ans =
    1794    6375
```

She sends (1794,6375) to Bob. Meanwhile, Bob calculates

```
>> multell([3,5],23,1,7206,7211)

ans =
    3861    1242
```

and sends (3861,1242) to Alice. Alice multiplies what she receives by NA and Bob multiplies what he receives by NB:

```
>>
multell([3861,1242],12,1,7206,7211)

ans =
    1472    2098

>>
multell([1794,6375],23,1,7206,7211)

ans =
    1472    2098
```

Therefore, Alice and Bob have produced the same key.

Appendix D Sage Examples

Sage is an open-source computer algebra package. It can be downloaded for free from www.sagemath.org/ or it can be accessed directly online at the website https://sagecell.sagemath.org/. The computer computations in this book can be done in Sage, especially by those comfortable with programming in Python. In the following, we give examples of how to do some of the basic computations. Much more is possible. See www.sagemath.org/ or search the Web for other examples. Another good place to start learning Sage in general is [Bard] (there is a free online version).

D.1 Computations for Chapter 2

Shift ciphers

Suppose you want to encrypt the plaintext `This is the plaintext` with a shift of 3. We first encode it as an alphabetic string of capital letters with the spaces removed. Then we shift each letter by three positions:

```
S=ShiftCryptosystem(AlphabeticStrings())
P=S.encoding("This is the
plaintext")
C=S.enciphering(3,P);C
```

When this is evaluated, we obtain the

ciphertext

```
WKLVLVWKHSODLQWHAW
```

To decrypt, we can shift by 23 or do the following:

```
S.deciphering(3,C)
```

When this is evaluated, we obtain

```
THISISTHEPLAINTEXT
```

Suppose we don’t know the key and we want to decrypt by trying all possible shifts:

```
S.brute_force(C)
```

Evaluation yields

```
0: WKLVLVWKHSODLQWHAW,
1: VJKUKUVJGRNCKPVGZV,
2: UIJTJTUIFQMBJOUFYU,
3: THISISTHEPLAINTEXT,
4: SGHRHRSGDOKZHMSDWS,
5: RFGQGQRFCNJYGLRCVR,
6: etc.

24: YMNXNXYMJUQFNSYJCY,
25: XLMWMWXLITPEMRXIBX
```

Affine ciphers

Let's encrypt the plaintext `This is the plaintext` using the affine function 3x+1 mod 26:

```
A=AffineCryptosystem(AlphabeticStrings())
P=A.encoding("This is the
plaintext")
```

```
C=A.enciphering(3,1,P);C
```

When this is evaluated, we obtain the ciphertext

```
GWZDZDGWNUIBZOGNSG
```

To decrypt, we can do the following:

```
A.deciphering(3,1,C)
```

When this is evaluated, we obtain

```
THISISTHEPLAINTEXT
```

We can also find the decryption key:

```
A.inverse_key(3,1)
```

This yields

```
(9, 17)
```

Of course, if we “encrypt” the ciphertext using 9x+17, we obtain the plaintext:

```
A.enciphering(9,17,C)
```

Evaluate to obtain

```
THISISTHEPLAINTEXT
```

Vigenère ciphers

Let’s encrypt the plaintext `This is the plaintext` using the keyword `ace` (that is, shifts of 0, 2, 4). Since we need to

express the keyword as an alphabetic string, it is efficient to add a symbol for these strings:

```
AS=AlphabeticStrings()
V=VigenereCryptosystem(AS,3)
K=AS.encoding("ace")
P=V.encoding("This is the
plaintext")
C=V.enciphering(K,P);C
```

The “3” in the expression for `V` is the length of the key. When the above is evaluated, we obtain the ciphertext

```
TJMSKWTJIPNEIPXEZX
```

To decrypt, we can shift by 0, 24, 22 (= ayw) or do the following:

```
```

```
V.deciphering(K,C)
```

When this is evaluated, we obtain

```
THISISTHEPLAINTEXT
```

Now let's try the example from Section 2.3. The ciphertext can be cut and pasted from ciphertexts.m in the MATLAB files (or, with a little more difficulty, from the Mathematica or Maple files). A few control symbols need to be removed in order to make the ciphertext a single string.

```
vvhq="vvhqwvvrhmusgjgthkihtssejchlsfcbgvwcrlry
. . . czvile"
```

(We omitted part of the ciphertext in the above in order to save space.) Now let's

compute the matches for various displacements. This is done by forming a string that displaces the ciphertext by i positions by adding i blank spaces at the beginning and then counting matches.

```
for i in range(0,7):
C2 = [" "]*i + list(C)
count = 0
for j in range(len(C)):
if C2[j] == C[j]:
count += 1
print i, count
```

The result is

```
0 331
1 14
2 14
3 16
4 14
5 24
6 12
```

The `331` is for a displacement of 0, so all 331 characters match. The high number of matches for a displacement of 5 suggests that the key length is 5. We now want to determine the key.

First, let's choose every fifth letter, starting with the first (counted as 0 for Sage). We extract these letters, put them in a list, then count the frequencies.

```
V1=list(C[0::5])
dict((x, V1.count(x)) for x in V1)
```

The result is

```
C: 7,
D: 1,
E: 1,
F: 2,
G: 9,
```

```
I: 1,
J: 8,
K: 8,
N: 3,
P: 4,
Q: 5,
R: 2,
T: 3,
U: 6,
V: 5,
W: 1,
Y: 1
```

Note that `A, B, H, L, M, O. S, X, Z` do not occur among the letters, hence are not listed. As discussed in Subsection 2.3.2 , the shift for these letters is probably 2. Now, let's choose every fifth letter, starting with the second (counted as 1 for Sage). We compute the frequencies:

```
V2=list(C[1::5])
```

```
dict((x, V2.count(x)) for x in V2)

A: 3,
B: 3,
C: 4,
F: 3,
G: 10,
H: 6,
M: 2,
O: 3,
P: 1,
Q: 2,
R: 3,
S: 12,
T: 3,
U: 2,
V: 3,
W: 3,
Y: 1,
Z: 2
```

As in Subsection 2.3.2 , the shift is probably 14. Continuing in this way, we find that the most likely key is {2, 14, 3, 4, 18}, which is `codes`. let's decrypt:

```
V=VigenereCryptosystem(AS,5)

K=AS.encoding("codes")
P=V.deciphering(K,C);P

THEMETHODUSEDFORTHEPREPARATIONANDREADINGOFCODE
. . . ALSETC
```

D.2 Computations for Chapter 3

To find the greatest common divisor, type the following first line and then evaluate:

```
gcd(119, 259)
7
```

To find the next prime greater than or equal to a number:

```
next_prime(1000)
1009
```

To factor an integer:

```
```

```
factor(2468)
2^2 * 617
```

Let’s solve the simultaneous congruences $x \equiv 1 \pmod 5$, $x \equiv 3 \pmod 7$:

```
crt(1,3,5,7)
31
```

To solve the three simultaneous congruences $x \equiv 1 \pmod 5$, $x \equiv 3 \pmod 7$, $x \equiv 0 \pmod{11}$:

```
a= crt(1,3,5,7)
crt(a,0,35,11)
66
```

Compute $123^{456} \pmod{789}$:

```
```

```
mod(123,789)456
699
```

Compute d so that $65d \equiv 1 \pmod{987}$:

```
mod(65,987)(-1)
410
```

Let's check the answer:

```
mod(65*410, 987)
1
```

D.3 Computations for Chapter 5

LFSR

Consider the recurrence relation xn +4 ≡ xn+xn+1+xn+3 mod 2, with initial values 1, 0, 0, 0. We need to use 0s and 1s, but we need to tell Sage that they are numbers mod 2. One way is to define "**o**" (that's a lower-case "oh") and "**l**" (that's an "ell") to be 0 and 1 mod 2:

```
F=GF(2)
o=F(0); l=F(1)
```

We also could use `F(0)` every time we want to enter a `0`, but the present

method saves some typing. Now we specify the coefficients and initial values of the recurrence relation, along with how many terms we want. In the following, we ask for 20 terms:

```
s=lfsr_sequence([1,1,0,1],
[1,0,0,0],20);s
```

This evaluates to

```
[1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1,
1, 1, 0, 0, 0, 1, 1, 1, 0]
```

Suppose we are given these terms of a sequence and we want to find what recurrence relation generates it:

```
berlekamp_massey(s)
```

This evaluates to

```
x^4 + x^3 + x + 1
```

When this is interpreted as x4 ≡ 1+1x +0x2+1x3 mod 2, we see that the coefficients 1, 1, 0, 1 of the polynomial give the coefficients of recurrence relation. In fact, it gives the smallest relation that generates the sequence.

Note: Even though the output for `s` has 0s and 1s, if we try entering the command `berlekamp_massey([1,1,0,1,1,0])` we get `x^3 - 1`. If we instead enter `berlekamp_massey([1,1,o,1,1,o])`, we get `x^2 + x + 1`. Why? The first is looking at a sequence of integers generated by the

relation $x_{n+3}=x_n$ while the second is looking at the sequence of integers mod 2 generated by $x_{n+2} \equiv x_n+x_{n+1} \bmod 2$. Sage defaults to integers if nothing is specified. But it remembers that the 0s and 1s that it wrote in **s** are still integers mod 2.

D.4 Computations for Chapter 6

Hill ciphers

Let's encrypt the plaintext `This is the plaintext` using a 3×3 matrix. First, we need to specify that we are working with such a matrix with entries in the integers mod 26:

```
R=IntegerModRing(26)
M=MatrixSpace(R,3,3)
```

Now we can specify the matrix that is the encryption key:

```

```

```
K=M([[1,2,3],[4,5,6],[7,8,10]]);K
```

Evaluate to obtain

```
[ 1 2  3]
[ 4 5  6]
[ 7 8 10]
```

This is the encryption matrix. We can now encrypt:

```
H=HillCryptosystem(AlphabeticStrings(),3)

P=H.encoding("This is the
plaintext")
C=H.enciphering(K,P);C
```

If the length of the plaintext is not a multiple of 3 (= the size of the matrix), then extra characters need to be appended to achieve this. When the

above is evaluated, we obtain the ciphertext

```
ZHXUMWXBJHHHLZGVPC
```

Decrypt:

```
H.deciphering(K,C)
```

When this is evaluated, we obtain

```
THISISTHEPLAINTEXT
```

We could also find the inverse of the encryption matrix mod 26:

```
K1=K.inverse();K1
```

This evaluates to

```
[ 8 16  1]
[ 8 21 24]
[ 1 24  1]
```

When we evaluate

```
H.enciphering(K,C):C
```

we obtain

```
THISISTHEPLAINTEXT
```

D.5 Computations for Chapter 9

Suppose someone unwisely chooses RSA primes p and q to be consecutive primes:

```
p=nextprime(987654321*10^50+12345);
q=nextprime(p+1)
n=p*q
```

Let's factor the modulus

n=pq=9754610577899710410000000000000000000000000000000000025076543210190000000000000000000000000000000000000000000000000161156941

without using the `factor` command:

```
s=N(sqrt(n), digits=70)
p1=next_prime(s)
```

```
p1, q1
(98765432100000000000000000000000000000000000

98765432100000000000000000000000000000000000
```

Of course, the fact that p and q are consecutive primes is important for this calculation to work. Note that we needed to specify 70-digit accuracy so that round-off error would not give us the wrong starting point for looking for the next prime. These factors we obtained match the original p and q, up to order:

```
p, q
(98765432100000000000000000000000000000000000

98765432100000000000000000000000000000000000
```

D.6 Computations for Chapter 10

Let's solve the discrete log problem $2x \equiv 71 \pmod{131}$ by the Baby Step-Giant Step method of Subsection 10.2.2. We take N=12 since N2 > p−1=130 and we form two lists. The first is $2j \pmod{1}31$ for $0 \leq j \leq 11$:

```
for i in range(0,12): print i,
mod(2,131)i

0 1
1 2
2 4
3 8
4 16
5 32
6 64
7 128
```

```
8 125
9 119
10 107
11 83
```

The second is $71 \cdot 2^{-j} \pmod{131}$ for $0 \le j \le 11$:

```
for i in range(0,12): print i,
mod(71*mod(2,131)(-12*i),131)

0 71
1 17
2 124
3 26
4 128
5 86
6 111
7 93
8 85
9 96
10 130
11 116
```

The number 128 is on both lists, so we

see that $27 \equiv 71 \cdot 2 - 12 * 4 \pmod{131}$. Therefore,

$$71 \equiv 27 + 4 * 12 \equiv 255 \pmod{131}.$$

D.7 Computations for Chapter 12

Suppose a lazy phone company employee assigns telephone numbers by choosing random seven-digit numbers. In a town with 10,000 phones, what is the probability that two people receive the same number?

```
i = var('i') 1-product(1.-
i/10^7,i,1,9999)
0.9932699132835016
```

D.8 Computations for Chapter 17

Lagrange interpolation

Suppose we want to find the polynomial of degree at most 3 that passes through the points (1, 1), (2, 2), (3, 21), (5, 12) mod the prime p=37. We first need to specify that we are working with polynomials in x mod 37. Then we compute the polynomial:

```
R=PolynomialRing(GF(37),"x")
f=R.lagrange_polynomial([(1,1),
(2,2),(3,21), (5,12)]);f
```

This evaluates to

```
22*x^3 + 25*x^2 + 31*x + 34
```

If we want the constant term:

```
f(0)
43
```

D.9 Computations for Chapter 18

Here is a game you can play. It is essentially the simplified version of poker over the telephone from Section 18.2.

There are five cards: ten, jack, queen, king, ace. They are shuffled and disguised by raising their numbers to a random exponent mod the prime `24691313099`. You are supposed to guess which one is the ace.

The cards are represented by `ten = 200514`, etc., because `t` is the 20th letter, `e` is the 5th letter, and `n` is the 14th

letter.

Type the following into Sage. The value of k forces the randomization to have 248 as its starting point, so the random choices of e and `shuffle` do not change when we repeat the process with this value of k. Varying k gives different results.

```
cards=[200514,10010311,1721050514,11091407,103

p=24691313099
k=248 set_random_seed(k)
e=randint(10,10^7)
def pow(list,ex):
   ret = []
   for i in list:
      ret.append(mod(i,p)^ex)
   return ret
s=pow(cards,2*e+1)
shuffle(s)
print(s)
```

Evaluation yields

```
[10426004161, 16230228497,
12470430058, 3576502017,
2676896936]
```

These are the five cards. None looks like the ace; that's because their numbers have been raised to powers mod the prime. Make a guess anyway. Let's see if you're correct.

Add the following line to the end of the program:

```
print(pow(s,mod(2*e+1,p-1)^(-1)))
```

and evaluate again:

```

```

```
[10426004161, 16230228497,
12470430058, 3576502017,
2676896936]
[10010311, 200514, 11091407, 10305,
1721050514]
```

The first line is the shuffled and hidden cards. The second line removed the kth power and reveals the cards. The fourth card is the ace. Were you lucky?

If you change the value of k, you can play again, but let's figure out how to cheat. Remove the last line of the program that you just added and replace it with

```
pow(s,(p-1)/2)
```

When the program is evaluated, we obtain

```
[10426004161, 16230228497,
12470430058, 3576502017,
2676896936]
[1, 1, 1, 24691313098, 1]
```

Why does this tell us the ace is in the fourth position? Read the part on "How to Cheat" in Section 18.2. Raise the numbers for the cards to the (p−1)/2 power mod p (you can put this extra line at the end of the program and ignore the previous output):

```
print(pow(cards,(p-1)/2))
[1, 1, 1, 1, 24691313098]
```

We see that the prime p was chosen so that only the ace is a quadratic nonresidue mod p.

If you input another value of k and play the game again, you'll have a similar situation, but with the cards in a different order.

D.10 Computations for Chapter 21

Elliptic curves

Let's set up the elliptic curve $y2 \equiv x3+2x+3 \bmod 7$:

```
E=EllipticCurve(IntegerModRing(7),
[2,3])
```

The entry `[2,3]` gives the coefficients a, b of the polynomial x3+ax+b. More generally, we could use the vector `[a,b,c,d,e]` to specify the coefficients of the general form $y2+axy +cy \equiv x3+bx2+dx+e$. We could also use `GF(7)` instead of `IntegerModRing(7)` to

specify that we are working mod 7. We could replace the `7` in `IntegerModRing(7)` with a composite modulus, but this will sometimes result in error messages when adding points (this is the basis of the elliptic curve factorization method).

We can list the points on E:

```
E.points()
[(0:1:0), (2:1:1), (2:6:1),
(3:1:1), (3:6:1), (6:0:1)]
```

These are given in projective form. The point `(0:1:0)` is the point at infinity. The point `(2:6:1)` can also be written as `[2, 6]`. We can add points:

```
E([2,1])+E([3,6])
(6 :  0 :  1)
```

```
E([0,1,0])+E([2,6])
(2 :  6 :  1)
```

In the second addition, we are adding the point at infinity to the point (2, 6) and obtaining (2, 6). This is an example of $\infty + P = P$. We can multiply a point by an integer:

```
5*E([2,1])
(2 :  6 : 1)
```

We can list the multiples of a point in a range:

```
for i in range(10):
   print(i,i*E([2,6]))

(0, (0 : 1 : 0))
(1, (2 : 6 : 1))
(2, (3 : 1 : 1))
```

```
(3, (6 : 0 : 1))
(4, (3 : 6 : 1))
(5, (2 : 1 : 1))
(6, (0 : 1 : 0))
(7, (2 : 6 : 1))
(8, (3 : 1 : 1))
(9, (6 : 0 : 1))
```

The indentation of the `print` line is necessary since it indicates that this is iterated by the `for` command. To count the number of points on E:

```
E.cardinality()
6
```

Sage has a very fast point counting algorithm (due to Atkins, Elkies, and Schoof; it is much more sophisticated than listing the points, which would be infeasible). For example,

```
p=next_prime(10^50)
E1=EllipticCurve(IntegerModRing(p),
[2,3])
n=E1.cardinality()
p, n, n-(p+1)
(100000000000000000000000000000000000000000000000

99999999999999999999999991123147331337610862 3

-88768526686623891376812 0)
```

As you can see, the number of points on this curve (the second output line) is close to p+1. In fact, as predicted by Hasse's theorem, the difference n−(p+1) (on the last output line) is less in absolute value than $2p \approx 2\times 1025$.

Appendix E Answers and Hints for Selected Odd-Numbered Exercises

Chapter 2

1. 1. Among the shifts of *EVIRE*, there are two words: *arena* and *river*. Therefore, Anthony cannot determine where to meet Caesar.

2. 3. The decrypted message is 'cat'.

3. 5. The ciphertext is *QZNHOBXZD*. The decryption function is 21x+9.

4. 7. The encryption function is therefore 11x+14.

5. 9. The plaintext is *happy*.

6. 11. Successively encrypting with two affine functions is the same as encrypting with a single affine function. There is therefore no advantage of doing double encryption in this case.

7. 13. Mod 27, there are 486 keys. Mod 29, there are 812 keys.

8. 15.

    1. The possible values for α are 1,7,11,13,17,19,23,29.

    2. There are many such possible answers, for example x=1 and x=4 will work. These correspond to the letters 'b' and 'e'.

9. 19.

    1. The key is *AB*. The original plaintext is *BBBBBBABBB*.

10. 21. The key is probably *AB*.

11. 27. *EK IO IR NO AN HF YG BZ YB LF GM ZN AG ND OD VC MK*

12. 29. AAXFFGDGAFAX

Chapter 3

1. 1.

1. One possibility: $1=(-1)\cdot 101+6\cdot 17$.

2. 6

2. 3.

1. $d=13$

2. Decryption is performed by raising the ciphertext to the 13th power mod 31.

3. 5.

1. $x \equiv 22 \pmod{59}$, or
   $x \equiv 22, 81, 140, 199 \pmod{236}$.

2. No solutions.

4. 7.

1. If $n=ab$ with $1<a, b<n$, then either $a \leq n$ of $b \leq n$. Otherwise, $ab>(n)(n)=n$. If $1<a< \leq n$, let p be a prime factor of a.

2. $\gcd(30030, 257)=1$.

3. No prime less than or equal to $257 \approx 16.03$ divides 257 because of the gcd calculation.

5. 9.

    1. The gcd is 257.

    2. $4883 = 257 \cdot 19$ and $4369 = 257 \cdot 17$.

6. 11.

    1. The gcd is 1.

    2. The gcd is 1.

    3. The gcd is 1.

7. 13.

    1. Use the Corollary in Section 3.2.

    2. Imitate the proof of the Corollary in Section 3.2.

8. 17. $x \equiv 23 \pmod{70}$.

9. 19. The smallest number is 58 and the next smallest number is 118.

10. 21.

    1. $x \equiv 43, 56, 87, 100 \pmod{143}$.

2. $x \equiv 44, 99 \pmod{143}$.

11. 23. The remainder is 8.

12. 25. The last two digits are 29.

13. 27. Use Fermat's theorem when $a \equiv 0 \pmod{p}$.

14. 29.

    1. $77 \equiv 3 \pmod{4}$.

    2. The last digit is 3.

15. 39.

    1. 521225.

    2. $b \equiv 0, 2, 4, 6, 8, 10, 12, 16, 18, 20, 22, 24 \pmod{26}$.

16. 41. 2 and 13

17. 43.

    1. No solutions.

    2. There are solutions.

    3. No solutions.

Chapter 4

1. 1.
    1. 0.
    2. P(M=cat|C=mxp) ≠ P(M=cat).
2. 3. The conditional probability is 0. Affine ciphers do not have perfect secrecy.
3. 5.
    1. 1/2.
    2. m0=HI, m1=BE is a possibility.
4. 11.
    1. Possible.
    2. Possible.
    3. Impossible.
5. 13. X

Chapter 5

1. 1.] The next four terms of the sequence are 1, 0, 0, 1.

2. 3. c0=1, c1=0, c2=0, c3=1.

3. 5. c0=2 and c1=1.

4. 7. kn+2 ≡ kn.

5. 9. c0=4 and c1=4.

Chapter 6

1. 1. *eureka.*

2. 3. 123112.

3. 5. 72135.

4. 7.

    1. 1091323.

    2. 10191319.

5. 9. Use *aabaab*.

6. 13.

    1. Alice’s method is more secure.

2. Compatibility with single encryption.

7. 19. The jth and the (j+1)st blocks.

Chapter 7

1. 1.

    1. Switch left and right halves and use the same procedure as encryption. Then switch the left and right of the final output.

    2. After two rounds, the ciphertext alone lets you determine M0 and therefore M1⊕K, but not M1 or K individually. If you also know the plaintext, you know M1 are therefore can deduce K.

    3. Three rounds is very insecure.

2. 3. The ciphertext from the second message can be decrypted to yield the password.

3. 5.

    1. The keys for each round are all 1s, so using them in reverse order doesn't

change anything.

2. All 0s.

4. 7. Show that when P^{-} and K^{-} are used, the input to the S-boxes is the same as when P and K are used.

Chapter 8

1. 1.

    1. We have $W(4)=W(0)\oplus T(W(0))=T(W(0))$. In the notation in Subsection 8.2.5, $a=b=c=d=0$. The S-box yields $e=f=g=h=99$ (base 10)$=01100011$ (binary). The round constant is $r(4)=000000100=00000001$. We have $e\oplus r(4)=01100100$. Therefore,

        $$W(4)=T(W(0))=01100100011000110110001101100011.$$

2. 3.

    1. Since addition in GF(28) is the same as $\oplus$, we have $f(x1)\oplus f(x2)=\alpha(x1+x2)=\alpha(x3+x4)=f(x3)\oplus f(x4)$.

Chapter 9

1. 1. The plaintext is 1415=no.
2. 3.
    1. d=27.
    2. Imitate the proof that RSA decryption works.
3. 5. The correct plaintext is 9.
4. 7. Use d=67.
5. 9. Solve $de \equiv 1 \pmod{p-1}$.
6. 11. Bob computes b1 with $bb1 \equiv 1 \pmod{\phi(n)}$ and raises e to the power b1.
7. 13. Divide the decryption by 2.
8. 15. It does not increase security.
9. 17. e=1 sends plaintext, and e=2 doesn't satisfy $\gcd(e, (p-1)(q-1))=1$.
10. 21. Compute a gcd.
11. 23. We have $(516107 \cdot 187722)2 \equiv (2 \cdot 7)2 \pmod{n}$.

12. 25. Combine the first three congruences. Ignore the fourth congruence.

13. 27. Use the Chinese Remainder Theorem to find x with $x \equiv 7 \pmod{p}$ and $x \equiv -7 \pmod{q}$.

14. 31. There are integers x and y such that xeA +yeB=1.

15. 33. *HELLO*

16. 41. 12345.

17. 45. Find d′ with $d'e \equiv 1 \pmod{12345}$.

18. 47.

    1. 1000000 messages.

Chapter 10

1. 1.

    1. L2(3)=4.

    2. $27 \equiv 11 \pmod{13}$.

2. 3.

1. 65 ≡ 10.

2. x is odd.

3. 5. x is even.

4. 7. (a), (b) L2(24)=72.

5. 9. x=122.

6. 13. Alice sends 10 to Bob and Bob sends 5 to Alice. Their shared secret is 6.

7. 15. Eve computes b1 with bb1 ≡ 1 (mod p−1).

Chapter 11

1. 1. It is easy to construct collisions: h(x)=h(x+p−1), for example. (However, it is fairly quickly computed (though not fast enough for real use), and it is preimage resistant.)

2. 3.

    1. Finding square roots mod pq is computationally equivalent to factoring.

    2. h(x) ≡ h(n−x) for all x

3. 5. It satisfies (1), but not (2) and (3).

4. 9. (a) and (b) Let h be either of the hash functions. Given y of length n, we have $h(y\|000\cdots)=y$.

5. 11. Collision resistance.

Chapter 12

1. 1. $165288 \approx .573$

2. 5. $1-e-7.3\times10-47 \approx 0$.

3. 7. It is very likely that two choose the same prime.

4. 9. The probability is approximately $1-e-500 \approx 0$ (or $1-e-450 \approx 0$ if you take numbers of exactly 15 digits).

Chapter 13

1. 1. Use the congruence defining s to solve for a.

2. 3. See Section 13.4.

3. 7.

    1. Let $m1 \equiv mr1r-1 \pmod{p-1}$.

4. 9. Imitate the proof for the usual ElGamal signatures.

5. 13. Eve notices that $\beta = r$.

Chapter 14

1. 1. Use the Birthday attack. Eve will probably factor some moduli.

2. 3.

    1. `0101` and `0110`.

3. 5.

    1. Enigma does not encrypt a letter to itself, so `DOG` is impossible.

    2. If the first of two long plaintexts is encrypted with Enigma, it is very likely that at least one letter of the second plaintext will match a letter of the ciphertext. More precisely, each individual letter of the second plaintext that doesn't match the first plaintext has probability around 1/26 of matching, so the probability is $1-(25/26)^{90} \approx 0.97$ that there is a match between the second

plaintext and the ciphertext. Therefore, Enigma does not have ciphertext indistinguishability.

Chapter 15

1. 1. KAB=gA(rB)=21, KAC=gA(rC)=7 and KBC=gB(rC)=29.

2. 3. a=8, b=8, c=23.

Chapter 16

1. 1.

    1. We have $r \equiv \alpha 1(cx+w)+\alpha 2 \equiv Hx+\alpha 1w+\alpha 2 \pmod q$. Therefore

        $gr \equiv gw\alpha 1g\alpha 2gxH \equiv gw\alpha 1g\alpha 2gxH \equiv ahH \pmod p$.

    2. Since $c1 \equiv w+xc \pmod q$, we have $\alpha 1c1 \equiv w\alpha 1+xH \pmod q$. Therefore,

        $r \equiv \alpha 1c1+\alpha 2 \equiv xH+w\alpha 1+\alpha 2 \pmod q$.

        Multiply by s and raise Ig2 to these exponents to obtain

(Ig2)rs ≡ (Ig2)xsH(Ig2)wsα1(Ig2)sα2 (mod p).

This may be rewritten as

Ar ≡ zHb (mod p).

3. Since r1 ≡ usd+x1 and r2 ≡ sd +x2 (mod q), we have

g1r1g2r2 ≡ (g1ug2)sdg1x1g2x2 ≡ (Ig2sdg1x1g2x2 ≡ ≡ A (mod p).

2. 3.

1. The only place r1 and r2 are used in the verification procedure is in checking that g1r1g2r2 ≡ AdB.

2. The Spender spends the coin correctly once, using r1, r2. The Spender then chooses any two random numbers r1′, r2′ with r1′+r2′=r1+r2 and uses the coin with the Vendor, with r1′, r2′ in place of r1, r2. All the verification equations work.

3. 5. r2 and r2′ are essentially random numbers (depending on hash values involving the clock), the probability is around 1/q that r2 ≡ r2′ (mod q). Since q is large, 1/q is small.

4. 7. Fred only needs to keep the hash of the file on his own computer.

Chapter 17

1. 1. One possibility is to take p=7 and choose the polynomial s(x)=5+x (mod 7) (where 5 is chosen randomly). Then the secret value is s(0)=5, and we may choose the shares (1,6), (2,0), (3,1), and (4,2).

2. 3. *=63

3. 5. The polynomial is

   $8(x-3)(x-5)(1-3)(1-5)+10(x-1)(x-5)(3-1)(3-5)+11(x-1)(x-3)(5-1)(5-3)$.

   The secret is 13 (mod 17).

4. 7. M=77, 57, 37, 17.

5. 9. Take a (10, 30) scheme and give the general 10 shares, the colonels five shares each, and the clerks two each.

6. 11. Start by splitting the launch code into three equal components using a three-party secret splitting scheme.

Chapter 19

1. 3.

    1. Nelson computes a square root of y mod p and mod q, then combines them to obtain a square root of y mod n.

    2. Use the $x2 \equiv y2$ factorization method.

    3. No.

2. 5.

    Step 4: Victor randomly chooses i=1 or 2 and asks Peggy for ri.

    Step 5: Victor checks that $xi \equiv rie \pmod{n}$.

    They repeat steps 1 through 5 at least 7 times (since $(1/2)7<.01$).

3. 7.

    1. One way: Step 4: Victor chooses $i \neq j$ at random and asks for ri and rj. Then five repetitions are enough. Another way: Victor asks for only one of the rk's. Then twelve repetitions suffice.

2. Choose r1, r2. Then solve for r3.

Chapter 20

1. 1. H(X1)=H(X2)=1, and H(X1, X2)=2.
2. 3. H(X)=2.
3. 5. H(Y) ≤ H(X).
4. 7.
    1. 2.9710
    2. 2.9517
5. 9.
    1. H(P)=−13log213+23log223.
    2. This system matches up with the one-time pad, and hence H(P|C)=H(P).
6. 13.
    1. H(X)=log236.
    2. Use Fermat's Theorem to obtain H(Y)=0.

Chapter 21

1. 3.

    1. (3, 2), (3, 5), (5, 2), (5, 5), (6, 2), (6, 5), ∞.

    2. (3, 5).

    3. (5, 2).

2. 5.

    1. (2, 2).

    2. She factors 35.

3. 7. One example: y2 ≡ x3+17.

4. 9.

    1. 3Q=(−1, 0).

5. 15. y ≡ ±4, ±11 (mod 35)

Chapter 22

1. 3. Compute e˜(aA, B) and e˜(A, bB).

2. 7.

    1. Eve knows rP0, P1, k. She computes

       e˜(rP0, P1)k=(˜kP0, P1)r=e˜(H1(bob@computer.com), P1

       Eve now computes H2(gr) and XORs it with t to get m.

3. 9. See Claim 2 in Section 9.1.

Chapter 23

1. 1. The basis (0, 1), (−2, 0) is reduced. The vector (0, 1) is a shortest vector.

Chapter 24

1. 1.

    1. The original message is 0,1,0,0.

    2. The original message is 0,1,0,1.

2. 3.

    1. n=5 and k=2.

2. G=1101010101.

3. (0, 0, 0, 0, 0), (1, 1, 0, 1, 0), (1, 0, 1, 0, 1), (0, 1, 1, 1,

4. R=log2(4)5=0.4.

3. 5.

    1. d(C)=2.

4. 13. 1+X+X2+X3 is in C and the other two polynomials are not in C.

5. 19. The error is in the 3rd position. The corrected vector is (1,0,0,1,0,1,1).

Chapter 25

1. 1.

    1. The period is 4.

    2. m=8.

    3. r=4.

Appendix F Suggestions for Further Reading

For the history of cryptography, see [Kahn] and [Bauer].

For additional treatment of topics in the present book, and many other topics, see [Stinson], [Stinson1], [Schneier], [Mao], and [Menezes et al.]. These books also have extensive bibliographies.

An approach emphasizing algebraic methods is given in [Koblitz].

For the theoretical foundations of cryptology, see [Goldreich1] and [Goldreich2]. See [Katz-Lindell] for an

approach based on security proofs.

Books that are oriented toward protocols and practical network security include [Stallings], [Kaufman et al.], and [Aumasson].

For a guidelines on properly applying cryptographic algorithms, the reader is directed to [Ferguson-Schneier]. For a general discussion on securing computing platforms, see [Pfleeger-Pfleeger].

The Internet, of course, contains a wealth of information about cryptographic issues. The Cryptology ePrint Archive server at `http://eprint.iacr.org/` contains very recent research. Also, the conference

proceedings CRYPTO, EUROCRYPT, and ASIACRYPT (published in Springer-Verlag's Lecture Notes in Computer Science series) contain many interesting reports on recent developments.

Bibliography

- [Adrian et al.] Adrian et al., "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice," `https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf`.

- [Agrawal et al.] M. Agrawal, N. Kayal, and N. Saxena, “PRIMES is in P,” Annals of Math. 160 (2004), 781–793.

- [Alford et al.] W. R. Alford, A. Granville, and C. Pomerance, “On the difficulty of finding reliable witnesses,” Algorithmic Number Theory, Lecture Notes in Computer Science 877, Springer-Verlag, 1994, pp. 1–16.

- [Alford et al. 2] W. R. Alford, A. Granville, and C. Pomerance, “There are infinitely many Carmichael numbers,” Annals of Math. 139 (1994), 703–722.

- [Atkins et al.] D. Atkins, M. Graff, A. Lenstra, P. Leyland, “The magic words are squeamish ossifrage,” Advances in Cryptology – ASIACRYPT ’94, Lecture Notes in Computer Science 917, Springer-Verlag, 1995, pp. 263–277.

- [Aumasson] J-P. Aumasson, Serious Cryptography: A Practical Introduction to Modern Encryption, No Starch Press, 2017.

- [Bard] G. Bard, Sage for Undergraduates, Amer. Math. Soc., 2015.

- [Bauer] C.Bauer, Secret History: The Story of Cryptology, CRC Press, 2013.

- [Beker-Piper] H. Beker and F. Piper, Cipher Systems: The Protection of Communications, Wiley-Interscience, 1982.

- [Bellare et al.] M. Bellare, R. Canetti, and H. Krawczyk, "Keying Hash Functions for Message Authentication," Advances in Cryptology (Crypto 96 Proceedings), Lecture Notes in Computer Science Vol. 1109, N. Koblitz ed., Springer-Verlag, 1996.

- [Bellare-Rogaway] M. Bellare and P. Rogaway, "Random oracles are practical: a paradigm for designing efficient protocols," First ACM Conference on Computer and Communications Security, ACM Press, New York, 1993, pp. 62–73.

- [Bellare-Rogaway2] M. Bellare and P. Rogaway, "Optimal asymmetric encryption," Advances in Cryptology – EUROCRYPT '94, Lecture Notes in

Computer Science 950, Springer-Verlag, 1995, pp. 92–111.

- [Berlekamp] E. Berlekamp, Algebraic Coding Theory, McGraw-Hill, 1968.

- [Bernstein et al.] Post-Quantum Cryptography, Bernstein, Daniel J., Buchmann, Johannes, Dahmen, Erik (Eds.), Springer-Verlag, 2009.

- [Bitcoin] bitcoin, `https://bitcoin.org/en/`

- [Blake et al.] I. Blake, G. Seroussi, N. Smart, Elliptic Curves in Cryptography, Cambridge University Press, 1999.

- [Blom] R. Blom, "An optimal class of symmetric key generation schemes," Advances in Cryptology – EUROCRYPT'84, Lecture Notes in Computer Science 209, Springer-Verlag, 1985, pp. 335–338.

- [Blum-Blum-Shub] L. Blum, M. Blum, and M. Shub, "A simple unpredictable pseudo-random number generator," SIAM Journal of Computing 15(2) (1986), 364–383.

- [Boneh] D. Boneh, "Twenty years of attacks on the RSA cryptosystem," Amer. Math. Soc. Notices 46 (1999), 203–213.

- [Boneh et al.] D. Boneh, G. Durfee, and Y. Frankel, “An attack on RSA given a fraction of the private key bits,” Advances in Cryptology – ASIACRYPT ’98, Lecture Notes in Computer Science 1514, Springer-Verlag, 1998, pp. 25–34.

- [Boneh-Franklin] D. Boneh and M. Franklin, “Identity based encryption from the Weil pairing,” Advances in Cryptology – CRYPTO ’01, Lecture Notes in Computer Science 2139, Springer-Verlag, 2001, pp. 213–229.

- [Boneh-Joux-Nguyen] D. Boneh, A. Joux, P. Nguyen, “Why textbook ElGamal and RSA encryption are insecure,” Advances in Cryptology – ASIACRYPT ’00, Lecture Notes in Computer Science 1976, Springer-Verlag, 2000, pp. 30–43.

- [Brands] S. Brands, “Untraceable off-line cash in wallets with observers,” Advances in Cryptology – CRYPTO’93, Lecture Notes in Computer Science 773, Springer-Verlag, 1994, pp. 302–318.

- [Campbell-Wiener] K. Campbell and M. Wiener, “DES is not a group,” Advances in Cryptology – CRYPTO ’92, Lecture Notes in Computer Science 740, Springer-Verlag, 1993, pp. 512–520.

- [Canetti et al.] R. Canetti, O. Goldreich, and S. Halevi, “The random oracle methodology,

revisited," Proceedings of the thirtieth annual ACM symposium on theory of computing, ACM Press, 1998, pp. 209–218.

- [Chabaud] F. Chabaud, "On the security of some cryptosystems based on error-correcting codes," Advances in Cryptology – EUROCRYPT'94, Lecture Notes in Computer Science 950, Springer-Verlag, 1995, pp. 131–139.

- [Chaum et al.] D. Chaum, E. van Heijst, and B. Pfitzmann, "Cryptographically strong undeniable signatures, unconditionally secure for the signer," Advances in Cryptology – CRYPTO '91, Lecture Notes in Computer Science 576, Springer-Verlag, 1992, pp. 470–484.

- [Cohen] H. Cohen, A Course in Computational Number Theory, Springer-Verlag, 1993.

- [Coppersmith1] D. Coppersmith, "The Data Encryption Standard (DES) and its strength against attacks," IBM Journal of Research and Development, vol. 38, no. 3, May 1994, pp. 243–250.

- [Coppersmith2] D. Coppersmith, "Small solutions to polynomial equations, and low exponent RSA vulnerabilities," J. Cryptology 10 (1997), 233–260.

- [Cover-Thomas] T. Cover and J. Thomas, Elements of Information Theory, Wiley Series in Telecommunications, 1991.

- [Crandall-Pomerance] R. Crandall and C. Pomerance, Prime Numbers, a Computational Perspective, Springer-Telos, 2000.

- [Crosby et al.] Crosby, S. A., Wallach, D. S., and Riedi, R. H. “Opportunities and limits of remote timing attacks,” ACM Trans. Inf. Syst. Secur. 12, 3, Article 17 (January 2009), 29 pages.

- [Damgård et al.] I. Damgård, P. Landrock, and C. Pomerance, “Average case error estimates for the strong probable prime test,” Mathematics of Computation 61 (1993), 177–194.

- [Dawson-Nielsen] E. Dawson and L. Nielsen, “Automated Cryptanalysis of XOR Plaintext Strings,” Cryptologia 20 (1996), 165–181.

- [Diffie-Hellman] W. Diffie and M. Hellman, “New directions in cryptography,” IEEE Trans. in Information Theory, 22 (1976), 644–654.

- [Diffie-Hellman2] W. Diffie and M. Hellman, “Exhaustive cryptanalysis of the NBS data encryption standard,” Computer 10(6) (June 1977), 74–84

- [Ekert-Josza] A. Ekert and R. Jozsa, “Quantum computation and Shor’s factoring algorithm,” Reviews of Modern Physics, 68 (1996), 733–753.

- [FIPS 186-2] FIPS 186-2, Digital signature standard (DSS), Federal Information Processing Standards Publication 186, U.S. Dept. of Commerce/National Institute of Standards and Technology, 2000.

- [FIPS 202] FIPS PUB 202, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, Federal Information Processing Standards Publication 202, U.S. Dept. of Commerce/National Institute of Standards and Technology, 2015, available at `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf`.

- [Ferguson-Schneier] N. Ferguson and B. Schneier, Practical Cryptography, Wiley, 2003.

- [Fortune-Merritt] S. Fortune and M. Merritt, “Poker Protocols,” Advances in Cryptology – CRYPTO’84, Lecture Notes in Computer Science 196, Springer-Verlag, 1985, pp. 454–464.

- [Gaines] H. Gaines, Cryptanalysis, Dover Publications, 1956.

- [Gallager] R. G. Gallager, Information Theory and Reliable Communication, Wiley, 1969.

- [Genkin et al.] D. Genkin, A. Shamir, and E. Tromer, “RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis,” December 18, 2013, available at `www.cs.tau.ac.il/~tromer/papers/acoustic-20131218.pdf`

- [Gilmore] Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design, Electronic Frontier Foundation, J. Gilmore (editor), O’Reilly and Associates, 1998.

- [Girault et al.] M. Girault, R. Cohen, and M. Campana, “A generalized birthday attack,” Advances in Cryptology – EUROCRYPT’88, Lecture Notes in Computer Science 330, Springer-Verlag, 1988, pp. 129–156.

- [Goldreich1] O. Goldreich, Foundations of Cryptography: Volume 1, Basic Tools, Cambridge University Press, 2001.

- [Goldreich2] O. Goldreich, Foundations of Cryptography: Volume 2, Basic Applications, Cambridge University Press, 2004.

- [Golomb] S. Golomb, Shift Register Sequences, 2nd ed., Aegean Park Press, 1982.

- [Hankerson et al.] D. Hankerson, A. Menezes, and S. Vanstone, Guide to Elliptic Curve Cryptography, Springer-Verlag, 2004.

- [Hardy-Wright] G. Hardy and E. Wright, An Introduction to the Theory of Numbers. Fifth edition, Oxford University Press, 1979.

- [Heninger et al.] N. Heninger, Z. Durumeric, E. Wustrow, J. A. Halderman, "Mining your Ps and Qs: Detection of widespread weak key in network devices," Proc. 21st USENIX Security Symposium, Aug. 2012; available at `https://factorable.net`.

- [HIP] R. Moskowitz and P. Nikander, "Host Identity Protocol (HIP) Architecture," May 2006; available at `https://tools.ietf.org/html/rfc4423`

- [Joux] A. Joux, "Multicollisions in iterated hash functions. Application to cascaded constructions," Advances in Cryptology – CRYPTO 2004, Lecture Notes in Computer Science 3152, Springer, 2004, pp. 306–316.

- [Kahn] D. Kahn, The Codebreakers, 2nd ed., Scribner, 1996.

- [Kaufman et al.] C. Kaufman, R. Perlman, M.

Speciner, Private Communication in a Public World. Second edition, Prentice Hall PTR, 2002.

- [Kilian-Rogaway] J. Kilian and P. Rogaway, “How to protect DES against exhaustive key search (an analysis of DESX),” J. Cryptology 14 (2001), 17–35.

- [Koblitz] N. Koblitz, Algebraic Aspects of Cryptography, Springer-Verlag, 1998.

- [Kocher] P. Kocher, “Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems,” Advances in Cryptology – CRYPTO ’96, Lecture Notes in Computer Science 1109, Springer, 1996, pp. 104–113.

- [Kocher et al.] P. Kocher, J. Jaffe, and B. Jun, “Differential power analysis,” Advances in Cryptology – CRYPTO ’99, Lecture Notes in Computer Science 1666, Springer, 1999, pp. 388–397.

- [Konikoff-Toplosky] J. Konikoff and S. Toplosky, “Analysis of Simplified DES Algorithms,” Cryptologia 34 (2010), 211–224.

- [Kozaczuk] W. Kozaczuk, Enigma: How the German Machine Cipher Was Broken, and How It Was Read by the Allies in World War Two; edited

and translated by Christopher Kasparek, Arms and Armour Press, London, 1984.

- [KraftW] J. Kraft and L. Washington, An Introduction to Number Theory with Cryptography, CRC Press, 2018.

- [Lenstra et al.] A. Lenstra, X. Wang, B. de Weger, “Colliding X.509 certificates,” preprint, 2005.

- [Lenstra2012 et al.] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter, “Ron was wrong, Whit is right,” `https://eprint.iacr.org/2012/064.pdf`.

- [Lin-Costello] S. Lin and D. J. Costello, Jr., Error Control Coding: Fundamentals and Applications, Prentice Hall, 1983.

- [MacWilliams-Sloane] F. J. MacWilliams and N. J. A. Sloane, The Theory of Error-Correcting Codes, North-Holland, 1977.

- [Mantin-Shamir] I. Mantin and A. Shamir, “A practical attack on broadcast RC4,” In: FSE 2001, 2001.

- [Mao] W. Mao, Modern Cryptography: Theory and Practice, Prentice Hall PTR, 2004.

- [Matsui] M. Matsui,“Linear cryptanalysis method for DES cipher,” Advances in Cryptology – EUROCRYPT’93, Lecture Notes in Computer Science 765, Springer-Verlag, 1994, pp. 386–397.

- [Menezes et al.] A. Menezes, P. van Oorschot, and S. Vanstone, Handbook of Applied Cryptography, CRC Press, 1997.

- [Merkle-Hellman] R. Merkle and M. Hellman, “On the security of multiple encryption,” Comm. of the ACM 24 (1981), 465–467.

- [Mikle] O. Mikle, “Practical Attacks on Digital Signatures Using MD5 Message Digest,” Cryptology ePrint Archive, Report 2004/356, `http://eprint.iacr.org/2004/356`, 2nd December 2004.

- [Nakamoto] S. Nakamoto, ”Bitcoin: A Peer-to-peer Electronic Cash System,” available at https://bitcoin.org/bitcoin.pdf

- [Narayanan et al.] A. Narayanan, J. Bonneau, E. Felten, A. Miller, S. Goldfeder, Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction (with a preface by Jeremy Clark), Princeton University Press 2016.

- [Nelson-Gailly] M. Nelson and J.-L. Gailly, The

Data Compression Book, M&T Books, 1996.

- [Nguyen-Stern] P. Nguyen and J. Stern, “The two faces of lattices in cryptology,” Cryptography and Lattices, International Conference, CaLC 2001, Lecture Notes in Computer Science 2146, Springer-Verlag, 2001, pp. 146–180.

- [Niven et al.] I. Niven, H. Zuckerman, and H. Montgomery, An Introduction to the Theory of Numbers, Fifth ed., John Wiley & Sons, Inc., New York, 1991.

- [Okamoto-Ohta] T. Okamoto and K. Ohta, “Universal electronic cash,” Advances in Cryptology – CRYPTO’91, Lecture Notes in Computer Science 576, Springer-Verlag, 1992, pp. 324–337.

- [Pfleeger-Pfleeger] C. Pfleeger, S. Pfleeger, Security in Computing. Third edition, Prentice Hall PTR, 2002.

- [Pomerance] C. Pomerance, “A tale of two sieves,” Notices Amer. Math. Soc. 43 (1996), no. 12, 1473–1485.

- [Quisquater et al.] J.-J. Quisquater and L. Guillou, “How to explain zero-knowledge protocols to your children,” Advances in Cryptology – CRYPTO

’89, Lecture Notes in Computer Science 435, Springer-Verlag, 1990, pp. 628–631.

- [Rieffel-Polak] E. Rieffel and W. Polak, “An Introduction to Quantum Computing for Non-Physicists,” available at xxx.lanl.gov/abs/quant-ph/9809016.

- [Rosen] K. Rosen, Elementary Number Theory and its Applications. Fourth edition, Addison-Wesley, Reading, MA, 2000.

- [Schneier] B. Schneier, Applied Cryptography, 2nd ed., John Wiley, 1996.

- [Shannon1] C. Shannon, “Communication theory of secrecy systems,” Bell Systems Technical Journal 28 (1949), 656–715.

- [Shannon2] C. Shannon, “A mathematical theory of communication,” Bell Systems Technical Journal, 27 (1948), 379–423, 623–656.

- [Shoup] V. Shoup, “OAEP Reconsidered,” CRYPTO 2001 (J. Killian (ed.)), Springer LNCS 2139, Springer-Verlag Berlin Heidelberg, 2001, pp. 239–259.

- [Stallings] W. Stallings, Cryptography and Network Security: Principles and Practice, 3rd ed.,

Prentice Hall, 2002.

- [Stevens et al.] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, Y. Markov, “The first collision for full SHA-1,” `https://shattered.io/static/shattered.pdf`.

- [Stinson] D. Stinson, Cryptography: Theory and Practice. Second edition, Chapman & Hall/CRC Press, 2002.

- [Stinson1] D. Stinson, Cryptography: Theory and Practice, CRC Press, 1995.

- [Thompson] T. Thompson, From Error-Correcting Codes through Sphere Packings to Simple Groups, Carus Mathematical Monographs, number 21, Mathematical Assoc. of America, 1983.

- [van der Lubbe] J. van der Lubbe, Basic Methods of Cryptography, Cambridge University Press, 1998.

- [van Oorschot-Wiener] P. van Oorschot and M. Wiener, “A known-plaintext attack on two-key triple encryption,” Advances in Cryptology – EUROCRYPT ’90, Lecture Notes in Computer Science 473, Springer-Verlag, 1991, pp. 318–325.

- [Wang et al.] X. Wang, D. Feng, X. Lai, H. Yu,

"Collisions for hash functions MD-4, MD-5, HAVAL-128, RIPEMD," preprint, 2004.

- [Wang et al. 2] X. Wang, Y. Yin, H. Yu, "Finding collisions in the full SHA1," to appear in CRYPTO 2005.

- [Washington] L. Washington, Elliptic Curves: Number Theory and Cryptography, Chapman & Hall/CRC Press, 2003.

- [Welsh] D. Welsh, Codes and Cryptography, Oxford, 1988.

- [Wicker] S. Wicker, Error Control Systems for Digital Communication and Storage, Prentice Hall, 1995.

- [Wiener] M. Wiener, "Cryptanalysis of short RSA secret exponents," IEEE Trans. Inform. Theory, 36 (1990), 553–558.

- [Williams] H. Williams, Edouard Lucas and Primality Testing, Wiley-Interscience, 1998.

- [Wu1] T. Wu, "The secure remote password protocol," In: Proc. of the Internet Society Network and Distributed Security Symposium, 97–111, March 1998.

- [Wu2] T. Wu, "SRP-6: Improvements and

refinements to the Secure Remote Password protocol," 2002; available through http://srp.stanford.edu/design.html

Index

- (n, M, d) code, 445
- GF(28), 73, 163
- GF(4), 69, 397, 479
- [n, k, d] code, 451
- Zp, 71
- $\oplus$, 108, 137
- ϕ(n), 57
- ar $\equiv$ 1 factorization method, 175, 192, 518, 544, 578
- p−1 method, 188, 393, 396, 519, 546, 580
- q-ary code, 442
- x2 $\equiv$ y2 factorization method, 494, 501
- G23, 463
- G24, 459
- 3DES, 155

- absorption, 238
- acoustic cryptanalysis, 183
- addition law, 384, 388, 404
- AddRoundKey, 161
- ADFGX cipher, 27
- Adleman, 171
- Advanced Encryption Standard (AES), 137, 160, 293
- Aesop, 37
- affine cipher, 12, 133, 380
- Agrawal, 188
- Alice, 2
- anonymity, 325, 329
- ASCII, 88
- asymptotic bounds, 449
- Athena, 299
- Atkins, 193

- attacks, 3
- attacks on RSA, 177, 426
- authenticated key agreement, 292
- authenticated key distribution, 295
- authentication, 8, 25, 196, 310, 314
- automatic teller machine, 357, 361

- baby step, giant step, 215, 249, 392
- basic principle, 55, 58, 184, 189
- basis, 421
- Batey, 283
- Bayes's theorem, 367
- BCH bound, 472
- BCH codes, 472
- Bellare, 180, 251, 255
- Berlekamp, 483
- Berson, 357
- Bertoni, 237
- Bidzos, 283
- Biham, 136, 140
- bilinear, 409
- bilinear Diffie-Hellman, 412

- bilinear pairing, 409
- binary, 88
- binary code, 442
- birthday attack, 246, 249, 250, 265, 274
- birthday paradox, 193, 246, 268, 284, 286
- bit, 88
- bit commitment, 218
- Bitcoin, 326, 330
- Blakley secret sharing scheme, 344, 348
- Bletchley Park, 33, 283
- blind signature, 271
- blind signature, restricted, 320, 325
- block cipher, 118, 137, 256
- block code, 443
- blockchains, 262, 328, 334
- Blom key pre-distribution scheme, 294

- BLS signatures, 414
- Blum-Blum-Shub, 106
- Bob, 2
- bombes, 33
- Boneh, 177, 412, 414, 417
- bounded storage, 90
- bounds on codes, 446
- Brands, 319, 320
- breaking DES, 152
- brute force attack, 6, 169
- burst errors, 480
- byte, 88

- Caesar cipher, 11
- Canetti, 253, 255
- Carmichael number, 80
- CBC-MAC, 256
- certificate, 303–307, 309, 312
- certification authority (CA), 303, 304
- certification hierarchy, 304
- certification path, 309
- CESG, 171, 195
- chain rule, 370
- challenge-response, 357
- characteristic 2, 396
- Chaum, 228, 271, 319
- cheating, 354
- check symbols, 453

- Chinese remainder theorem, 52, 53, 84, 209
- chosen ciphertext attack, 3
- chosen plaintext attack, 3
- CI Game, 97, 252, 289
- cipher block chaining (CBC), 119, 123, 134, 256
- cipher feedback (CFB), 119, 123, 134, 168
- ciphers, 5
- ciphertext, 2
- ciphertext indistinguishability, 97, 181, 207, 289
- ciphertext only attack, 3
- Cliff, 299
- closest vector problem, 434, 435
- Cocks, 171, 195
- code, 442
- code rate, 440, 445, 446
- codes, 5

- codeword, 437, 443
- coding gain, 442
- coding theory, 1, 437
- coin, 321
- collision, 227, 228
- collision resistant, 226, 229
- composite, 41
- compression function, 231, 234, 249
- computational Diffie-Hellman, 220, 222, 412
- computationally infeasible, 195, 196
- conditional entropy, 370
- conditional probability, 94, 367
- confidentiality, 8
- confusion, 119
- congruence, 47
- continued fractions, 76, 82, 83, 178, 500

- convolutional codes, 483
- Coppersmith, 149, 151, 177
- correct errors, 444
- coset, 455
- coset leader, 455, 456
- counter mode (CTR), 128
- CRC-32, 287
- cryptanalysis, 1
- cryptocurrencies, 329
- cryptography, 1
- cryptology, 1
- cyclic codes, 466

- Daemen, 160, 237
- Damgård, 231
- Data Encryption Standard (DES), 131, 136, 145, 156
- Daum, 232
- decision Diffie-Hellman, 220, 222, 420
- decode, 442
- DES Challenge, 153
- DES Cracker, 153
- DESX, 129
- detect errors, 444
- deterministic, 249
- Di Crescenzo, 417
- dictionary attack, 155
- differential cryptanalysis, 140, 168

- Diffie, 152, 171, 195, 292
- Diffie-Hellman, 220, 222, 411, 420
- Diffie-Hellman key exchange, 219, 291, 401, 526, 554, 590
- diffusion, 118, 168, 169
- digital cash, 320
- digital signature, 8, 292, 401
- Digital Signature Algorithm (DSA), 215, 275, 402, 406
- digram, 21, 27, 121, 376
- Ding, 90
- discrete logarithm, 74, 84, 211, 228, 249, 272, 324, 352, 361, 363, 391, 399, 410
- Disparition, La, 15
- divides, 40
- dot product, 18, 35, 442, 453, 456, 489
- double encryption, 129, 130, 149, 198, 203
- dual code, 456, 485

• dual signature, 315

- electronic cash, 9
- electronic codebook (ECB), 119, 122
- Electronic Frontier Foundation, 153
- electronic voting, 207
- ElGamal cryptosystem, 196, 221, 400, 525, 553, 589
- ElGamal signature, 271, 276, 278, 279, 401, 407
- elliptic curve cryptosystems, 399
- elliptic curves, 189, 384
- elliptic integral, 386
- Ellis, 171
- encode, 442
- Enigma, 29, 282
- entropy, 367, 368
- entropy of English, 376

- entropy rate, 382
- equivalent codes, 445
- error correcting codes, 437, 442
- error correction, 26
- error propagation, 119
- Euclidean algorithm, 43, 82, 85
- Euler's ϕ-function, 57, 81, 174
- Euler's theorem, 57, 173
- Eve, 2
- everlasting security, 90
- existential forgery, 278
- expansion permutation, 145
- extended Euclidean algorithm, 44, 72

- factor base, 190, 216
- factoring, 188, 191, 212, 393, 494
- factorization records, 191, 212
- Feige-Fiat-Shamir identification, 359
- Feistel system, 137, 138, 168
- Feng, 227
- Fermat factorization, 188
- Fermat prime, 82
- Fermat's theorem, 55, 184
- Feynman, 488
- Fibonacci numbers, 78
- field, 70
- finite field, 69, 163, 396, 397, 468
- Flame, 227
- flipping coins, 349

- football, 218
- Fourier transform, 495, 499, 502
- fractions, 51
- Franklin, 412
- fraud control, 324
- frequencies of letters, 14, 21
- frequency analysis, 21

- Gadsby, 14
- games, 9, 349
- generating matrix, 452
- generating polynomial, 468
- Gentry, 206
- Gilbert-Varshamov bound, 448, 450, 485
- Golay code, 459
- Goldberg, 283
- Goldreich, 253
- Goppa codes, 449, 481
- Graff, 193
- Grant, 300
- greatest common divisor (gcd), 42, 85
- group, 149
- Guillou, 357

- Hadamard code, 441, 450
- Halevi, 253
- Hamming bound, 447
- Hamming code, 439, 450, 457, 485
- Hamming distance, 443
- Hamming sphere, 446
- Hamming weight, 452
- hash function, 226–228, 230, 231, 233, 251, 253, 273, 312, 315, 336, 361
- hash pointer, 262
- Hasse's theorem, 391, 407, 408
- Hellman, 129, 152, 171, 195, 213
- Hess, 415
- hexadecimal, 234
- Hill cipher, 119

- HMAC, 255
- Holmes, Sherlock, 23
- homomorphic encryption, 206
- hot line, 90
- Huffman codes, 371, 378

- IBM, 136, 160
- ID-based signatures, 415
- identification scheme, 9, 359
- identity-based encryption, 412
- independent, 94, 366
- index calculus, 216, 391, 399
- indistinguishability, 252
- infinity, 385
- information rate, 445
- information symbols, 453
- information theory, 365
- initial permutation, 145, 149
- integrity, 8, 228, 314
- intruder-in-the-middle, 59, 290, 291, 317
- inverting matrices, 61

- irreducible polynomial, 71
- ISBN, 440, 484
- IV, 123, 126, 231, 249, 256, 285

- Jacobi symbol, 64, 66, 187
- Joux, 249, 411

- Kayal, 188
- Keccak, 237
- Kerberos, 299
- Kerckhoffs's principle, 4
- ket, 489
- key, 2
- key agreement, 293
- key distribution, 293, 491
- key establishment, 9
- key exchange, 401
- key length, 5, 16, 384, 482
- key permutation, 148
- key pre-distribution, 294
- key schedule, 165, 169
- keyring, 309

- keyword search, 417
- knapsack problem, 195
- known plaintext attack, 3
- Koblitz, 384, 392
- Kocher, 181
- Krawczyk, 255

- Lagrange interpolation, 341, 343
- Lai, 227
- Lamport, 260
- lattice, 421
- lattice reduction, 422
- ledger, 328, 331
- Legendre symbol, 64, 82
- length, 107
- length extension attack, 232, 255
- Lenstra, A., 193, 227, 284, 425
- Lenstra, H. W., 384, 425
- Leyland, 193
- linear code, 451
- linear congruential generator, 105
- linear cryptanalysis, 144, 168

- linear feedback shift register (LFSR), 74, 107
- LLL algorithm, 425, 427
- Lovász, L., 425
- LUCIFER, 137
- Luks, 232
- Lynn, 414

- MAC, 255, 313
- Mantin, 114
- Maple, 527
- Mariner, 441
- MARS, 160
- Massey, 59, 483
- Mathematica, 503
- MATLAB, 555
- matrices, 61
- Matsui, 144
- Mauborgne, 88
- Maurer, 90
- McEliece cryptosystem, 197, 435, 480
- MD5, 227, 233
- MDS code, 446, 450

- meet-in-the-middle attack, 129, 130, 133
- Menezes, 400, 410
- Merkle, 129, 231
- Merkle tree, 263, 337
- Merkle-Damgård, 231
- message authentication code (MAC), 255, 313
- message digest, 226
- message recovery scheme, 273
- Mikle, 227
- Miller, 384
- Miller-Rabin primality test, 185, 209
- minimum distance, 444
- mining, 327
- MIT, 299
- MixColumns transformation, 161, 164, 169
- mod, 47

- modes of operation, 122
- modular exponentiation, 54, 84, 182, 276
- Morse code, 372
- MOV attack, 410
- multicollision, 249, 253, 268
- multiple encryption, 129
- multiplicative inverse, 49, 73, 165

- Nakamoto, 330
- National Bureau of Standards (NBS), 136, 152
- National Institute of Standards and Technology (NIST), 136, 152, 160, 233, 384, 397
- National Security Agency (NSA), 37, 136, 152, 233
- nearest neighbor decoding, 444
- Needham–Schroeder protocol, 297
- Netscape, 283
- Newton interpolating polynomial, 348
- NIST, 237
- non-repudiation, 8, 196
- nonce, 296, 327, 337, 339
- NP-complete, 456
- NTRU, 197, 429
- number field sieve, 191

- OAEP, 180, 252
- Ohta, 318
- Okamoto, 318, 410
- Omura, 59
- one-time pad, 88, 89, 91, 97, 252, 282, 286, 375, 380
- one-way function, 105, 155, 196, 212, 219, 226
- order, 83, 404
- Ostrovsky, 417
- output feedback (OFB), 126

- padding, 180, 235, 238, 255
- Paillier cryptosystem, 206
- Painvin, 28
- pairing, 409
- parity check, 438
- parity check matrix, 453, 470
- passwords, 155, 224, 256, 258, 260
- Peeters, 237
- Peggy, 357
- Pell's equation, 83
- perfect code, 447, 485, 486
- perfect secrecy, 95, 373, 374
- Persiano, 417
- Pfitzmann, 228
- plaintext, 2, 392

- plaintext-aware encryption, 181
- Playfair cipher, 26
- Pohlig-Hellman algorithm, 213, 224, 276, 391, 407
- point at infinity, 385
- poker, 351
- polarization, 489
- Pollard, 188
- post-quantum cryptography, 435
- PostScript, 232
- preimage resistant, 226
- Pretty Good Privacy (PGP), 309
- PRGA, 114
- primality testing, 183
- prime, 41
- prime number theorem, 41, 185, 245, 279
- primitive root, 59, 82, 83

- primitive root of unity, 472
- probabilistic, 249
- probability, 365
- provable security, 94
- pseudoprime, 185
- pseudorandom, 238, 284
- pseudorandom bits, 105
- public key cryptography, 4, 171, 195
- Public Key Infrastructure (PKI), 303

- quadratic reciprocity, 67
- quadratic residue, 354, 355
- quadratic residuosity problem, 69
- quadratic sieve, 205
- quantum computing, 493
- quantum cryptography, 488, 491
- quantum Fourier transform, 499
- qubit, 491
- Quisquater, 357

- R Game, 98, 114
- Róż˙ycki, 29
- Rabin, 84, 90
- random oracle model, 251
- random variable, 366
- RC4, 113, 285
- RC6, 160
- recurrence relation, 74, 107
- reduced basis, 422
- redundancy, 379
- Reed-Solomon codes, 479
- registration authority (RA), 304
- Rejewski, 29, 32
- relatively prime, 42
- repetition code, 437, 450

- restricted blind signature, 320, 325
- Rijmen, 160
- Rijndael, 160
- Rivest, 113, 129, 171, 233
- Rogaway, 180, 251
- root of unity, 472
- rotation, 230
- rotor machines, 29
- round constant, 165, 169
- round key, 165
- RoundKey addition, 164
- RSA, 97, 171, 172, 174, 222, 225, 283, 293, 376, 426, 433
- RSA challenge, 192
- RSA signature, 194, 270, 310
- run length coding, 381

- S-box, 139, 148–150, 163, 165, 168
- Safavi-Naini, 415
- Sage, 591
- salt, 155, 258
- Saxena, 188
- Schacham, 414
- Scherbius, 29
- Schnorr identification scheme, 363
- secret sharing, 9, 340
- secret splitting, 340
- Secure Electronic Transaction (SET), 314
- Secure Hash Algorithm (SHA), 227, 233, 235
- Secure Remote Password (SRP) protocol, 258
- Security Sockets Layer (SSL), 312
- seed, 98, 105

- self-dual code, 457
- sequence numbers, 296
- Serge, 299
- Serpent, 160
- SHA-3, 237
- SHAKE, 238
- Shamir, 59, 114, 136, 140, 171, 412
- Shamir threshold scheme, 341
- Shannon, 118, 365, 368, 371, 377, 378
- shift cipher, 11, 97
- ShiftRows transformation, 161, 164, 169
- Shor, 488, 493
- Shor's algorithm, 493, 497
- shortest vector, 424, 425
- shortest vector problem, 422
- side-channel attacks, 183

- signature with appendix, 273
- Singleton bound, 446
- singular curves, 395
- smooth, 394
- Solovay-Strassen, 187
- sphere packing bound, 447
- sponge function, 237
- square roots, 53, 62, 85, 218, 349, 358, 362
- squeamish ossifrage, 194
- squeezing, 238
- state, 237
- station-to-station (STS) protocol, 292
- stream cipher, 104
- strong pseudoprime, 185, 209
- strongly collision resistant, 226
- SubBytes transformation, 161, 163

- substitution cipher, 20, 380
- supersingular, 410, 419
- Susilo, 415
- Sybil attack, 338
- symmetric key, 4, 196
- syndrome, 455, 456
- syndrome decoding, 456
- systematic code, 453

- ternary code, 442
- three-pass protocol, 58, 198, 282, 317
- threshold scheme, 341
- ticket-granting service, 300
- timestamps, 296
- timing attacks, 181
- Transmission Control Protocol (TCP), 483
- Transport Layer Security (TLS), 312
- trapdoor, 136, 196, 197
- treaty verification, 194
- Trent, 300
- triangle inequality, 444
- trigram, 121, 376
- tripartite Diffie-Hellman, 411
- triple encryption, 129, 131

- trust, 304, 309
- trusted authority, 292, 294, 295, 300, 361
- Turing, 33
- two lists, 180
- two-dimensional parity code, 438
- Twofish, 160

- unicity distance, 379
- Unix, 156

- Van Assche, 237
- van Heijst, 228
- van Oorschot, 292
- Vandermonde determinant, 342, 473
- Vanstone, 400, 410
- variance, 182
- Vernam, 88
- Verser, 153
- Victor, 357
- Vigenère cipher, 14
- Void, A, 15
- Voyager, 459

- Wagner, 283
- Wang, 227
- weak key, 151, 158, 159, 168
- web of trust, 309
- Weil pairing, 410
- WEP, 284
- Wiener, 152, 178, 292
- World War I, 5, 26
- World War II, 3, 29, 33, 294
- WPA, 284

- X.509, 227, 245, 284, 304–307, 312
- XOR, 89, 137

- Yin, 227
- Yu, 227

- zero-knowledge, 357
- Zhang, 415
- Zimmerman, 309
- Zygalski, 29

Contents

1. Introduction to Cryptography with Coding Theory
2. Contents
3. Preface
4. Chapter 1 Overview of Cryptography and Its Applications
    1. 1.1 Secure Communications
        1. 1.1.1 Possible Attacks
        2. 1.1.2 Symmetric and Public Key Algorithms
        3. 1.1.3 Key Length
    2. 1.2 Cryptographic Applications
5. Chapter 2 Classical Cryptosystems
    1. 2.1 Shift Ciphers
    2. 2.2 Affine Ciphers
    3. 2.3 The Vigenère Cipher
        1. 2.3.1 Finding the Key Length
        2. 2.3.2 Finding the Key: First Method
        3. 2.3.3 Finding the Key: Second

4. 2.4 Substitution Ciphers
5. 2.5 Sherlock Holmes
6. 2.6 The Playfair and ADFGX Ciphers
7. 2.7 Enigma
8. 2.8 Exercises
9. 2.9 Computer Problems

6. Chapter 3 Basic Number Theory

1. 3.1 Basic Notions

    1. 3.1.1 Divisibility
    2. 3.1.2 Prime Numbers
    3. 3.1.3 Greatest Common Divisor

2. 3.2 The Extended Euclidean Algorithm
3. 3.3 Congruences

    1. 3.3.1 Division
    2. 3.3.2 Working with Fractions

4. 3.4 The Chinese Remainder Theorem
5. 3.5 Modular Exponentiation
6. 3.6 Fermat's Theorem and Euler's Theorem

    1. 3.6.1 Three-Pass Protocol

7. 3.7 Primitive Roots

8. 3.8 Inverting Matrices Mod n
9. 3.9 Square Roots Mod n
10. 3.10 Legendre and Jacobi Symbols
11. 3.11 Finite Fields
    1. 3.11.1 Division
    2. 3.11.2 $GF(2^8)$
    3. 3.11.3 LFSR Sequences
12. 3.12 Continued Fractions
13. 3.13 Exercises
14. 3.14 Computer Problems

7. Chapter 4 The One-Time Pad
    1. 4.1 Binary Numbers and ASCII
    2. 4.2 One-Time Pads
    3. 4.3 Multiple Use of a One-Time Pad
    4. 4.4 Perfect Secrecy of the One-Time Pad
    5. 4.5 Indistinguishability and Security
    6. 4.6 Exercises

8. Chapter 5 Stream Ciphers
    1. 5.1 Pseudorandom Bit Generation
    2. 5.2 Linear Feedback Shift Register Sequences
    3. 5.3 RC4
    4. 5.4 Exercises
    5. 5.5 Computer Problems

9. Chapter 6 Block Ciphers

    1. 6.1 Block Ciphers
    2. 6.2 Hill Ciphers
    3. 6.3 Modes of Operation

        1. 6.3.1 Electronic Codebook (ECB)
        2. 6.3.2 Cipher Block Chaining (CBC)
        3. 6.3.3 Cipher Feedback (CFB)
        4. 6.3.4 Output Feedback (OFB)
        5. 6.3.5 Counter (CTR)

    4. 6.4 Multiple Encryption
    5. 6.5 Meet-in-the-Middle Attacks
    6. 6.6 Exercises
    7. 6.7 Computer Problems

10. Chapter 7 The Data Encryption Standard

    1. 7.1 Introduction
    2. 7.2 A Simplified DES-Type Algorithm
    3. 7.3 Differential Cryptanalysis

        1. 7.3.1 Differential Cryptanalysis for Three Rounds
        2. 7.3.2 Differential Cryptanalysis for Four Rounds

    4. 7.4 DES

1. 7.4.1 DES Is Not a Group

5. 7.5 Breaking DES
6. 7.6 Password Security
7. 7.7 Exercises
8. 7.8 Computer Problems

11. Chapter 8 The Advanced Encryption Standard: Rijndael

1. 8.1 The Basic Algorithm
2. 8.2 The Layers

1. 8.2.1 The SubBytes Transformation
2. 8.2.2 The ShiftRows Transformation
3. 8.2.3 The MixColumns Transformation
4. 8.2.4 The RoundKey Addition
5. 8.2.5 The Key Schedule
6. 8.2.6 The Construction of the S-Box

3. 8.3 Decryption
4. 8.4 Design Considerations
5. 8.5 Exercises

12. Chapter 9 The RSA Algorithm

1. 9.1 The RSA Algorithm
2. 9.2 Attacks on RSA

1. 9.2.1 Low Exponent Attacks
2. 9.2.2 Short Plaintext
3. 9.2.3 Timing Attacks

3. 9.3 Primality Testing
4. 9.4 Factoring

1. 9.4.1 x2 ≡ y2
2. 9.4.2 Using ar ≡ 1

5. 9.5 The RSA Challenge
6. 9.6 An Application to Treaty Verification
7. 9.7 The Public Key Concept
8. 9.8 Exercises
9. 9.9 Computer Problems

13. Chapter 10 Discrete Logarithms

1. 10.1 Discrete Logarithms
2. 10.2 Computing Discrete Logs

1. 10.2.1 The Pohlig-Hellman Algorithm
2. 10.2.2 Baby Step, Giant Step
3. 10.2.3 The Index Calculus
4. 10.2.4 Computing Discrete Logs Mod 4

3. 10.3 Bit Commitment
4. 10.4 Diffie-Hellman Key Exchange
5. 10.5 The ElGamal Public Key Cryptosystem

1. 10.5.1 Security of ElGamal Ciphertexts

6. 10.6 Exercises
7. 10.7 Computer Problems

14. Chapter 11 Hash Functions

1. 11.1 Hash Functions
2. 11.2 Simple Hash Examples
3. 11.3 The Merkle-Damgård Construction
4. 11.4 SHA-2

1. Padding and Preprocessing
2. The Algorithm

5. 11.5 SHA-3/Keccak
6. 11.6 Exercises

15. Chapter 12 Hash Functions: Attacks and Applications

1. 12.1 Birthday Attacks

1. 12.1.1 A Birthday Attack on Discrete Logarithms

2. 12.2 Multicollisions
3. 12.3 The Random Oracle Model
4. 12.4 Using Hash Functions to Encrypt
5. 12.5 Message Authentication Codes

        1. 12.5.1 HMAC
        2. 12.5.2 CBC-MAC

    6. 12.6 Password Protocols

        1. 12.6.1 The Secure Remote Password protocol
        2. 12.6.2 Lamport's protocol

    7. 12.7 Blockchains
    8. 12.8 Exercises
    9. 12.9 Computer Problems

16. Chapter 13 Digital Signatures

    1. 13.1 RSA Signatures
    2. 13.2 The ElGamal Signature Scheme
    3. 13.3 Hashing and Signing
    4. 13.4 Birthday Attacks on Signatures
    5. 13.5 The Digital Signature Algorithm
    6. 13.6 Exercises
    7. 13.7 Computer Problems

17. Chapter 14 What Can Go Wrong

    1. 14.1 An Enigma "Feature"
    2. 14.2 Choosing Primes for RSA
    3. 14.3 WEP

        1. 14.3.1 CRC-32

4. 14.4 Exercises

18. Chapter 15 Security Protocols

    1. 15.1 Intruders-in-the-Middle and Impostors

        1. 15.1.1 Intruder-in-the-Middle Attacks

    2. 15.2 Key Distribution

        1. 15.2.1 Key Pre-distribution
        2. 15.2.2 Authenticated Key Distribution

    3. 15.3 Kerberos
    4. 15.4 Public Key Infrastructures (PKI)
    5. 15.5 X.509 Certificates
    6. 15.6 Pretty Good Privacy
    7. 15.7 SSL and TLS
    8. 15.8 Secure Electronic Transaction
    9. 15.9 Exercises

19. Chapter 16 Digital Cash

    1. 16.1 Setting the Stage for Digital Economies
    2. 16.2 A Digital Cash System

        1. 16.2.1 Participants
        2. 16.2.2 Initialization
        3. 16.2.3 The Bank

4. 16.2.4 The Spender
5. 16.2.5 The Merchant
6. 16.2.6 Creating a Coin
7. 16.2.7 Spending the Coin
8. 16.2.8 The Merchant Deposits the Coin in the Bank
9. 16.2.9 Fraud Control
10. 16.2.10 Anonymity

3. 16.3 Bitcoin Overview

1. 16.3.1 Some More Details

4. 16.4 Cryptocurrencies
5. 16.5 Exercises

20. Chapter 17 Secret Sharing Schemes

1. 17.1 Secret Splitting
2. 17.2 Threshold Schemes
3. 17.3 Exercises
4. 17.4 Computer Problems

21. Chapter 18 Games

1. 18.1 Flipping Coins over the Telephone
2. 18.2 Poker over the Telephone

1. 18.2.1 How to Cheat

3. 18.3 Exercises

22. Chapter 19 Zero-Knowledge Techniques

    1. 19.1 The Basic Setup
    2. 19.2 The Feige-Fiat-Shamir Identification Scheme
    3. 19.3 Exercises

23. Chapter 20 Information Theory

    1. 20.1 Probability Review
    2. 20.2 Entropy
    3. 20.3 Huffman Codes
    4. 20.4 Perfect Secrecy
    5. 20.5 The Entropy of English

        1. 20.5.1 Unicity Distance

    6. 20.6 Exercises

24. Chapter 21 Elliptic Curves

    1. 21.1 The Addition Law
    2. 21.2 Elliptic Curves Mod p

        1. 21.2.1 Number of Points Mod p
        2. 21.2.2 Discrete Logarithms on Elliptic Curves
        3. 21.2.3 Representing Plaintext

3. 21.3 Factoring with Elliptic Curves
    1. 21.3.1 Singular Curves
4. 21.4 Elliptic Curves in Characteristic 2
5. 21.5 Elliptic Curve Cryptosystems
    1. 21.5.1 An Elliptic Curve ElGamal Cryptosystem
    2. 21.5.2 Elliptic Curve Diffie-Hellman Key Exchange
    3. 21.5.3 ElGamal Digital Signatures
6. 21.6 Exercises
7. 21.7 Computer Problems

25. Chapter 22 Pairing-Based Cryptography
    1. 22.1 Bilinear Pairings
    2. 22.2 The MOV Attack
    3. 22.3 Tripartite Diffie-Hellman
    4. 22.4 Identity-Based Encryption
    5. 22.5 Signatures
        1. 22.5.1 BLS Signatures
        2. 22.5.2 A Variation
        3. 22.5.3 Identity-Based Signatures
    6. 22.6 Keyword Search
    7. 22.7 Exercises

26. Chapter 23 Lattice Methods

    1. 23.1 Lattices
    2. 23.2 Lattice Reduction

        1. 23.2.1 Two-Dimensional Lattices
        2. 23.2.2 The LLL algorithm

    3. 23.3 An Attack on RSA
    4. 23.4 NTRU

        1. 23.4.1 An Attack on NTRU

    5. 23.5 Another Lattice-Based Cryptosystem
    6. 23.6 Post-Quantum Cryptography?
    7. 23.7 Exercises

27. Chapter 24 Error Correcting Codes

    1. 24.1 Introduction
    2. 24.2 Error Correcting Codes
    3. 24.3 Bounds on General Codes

        1. 24.3.1 Upper Bounds
        2. 24.3.2 Lower Bounds

    4. 24.4 Linear Codes

        1. 24.4.1 Dual Codes

5. 24.5 Hamming Codes
6. 24.6 Golay Codes
    1. Decoding G24
7. 24.7 Cyclic Codes
8. 24.8 BCH Codes
    1. 24.8.1 Decoding BCH Codes
9. 24.9 Reed-Solomon Codes
10. 24.10 The McEliece Cryptosystem
11. 24.11 Other Topics
12. 24.12 Exercises
13. 24.13 Computer Problems

28. Chapter 25 Quantum Techniques in Cryptography

1. 25.1 A Quantum Experiment
2. 25.2 Quantum Key Distribution
3. 25.3 Shor's Algorithm
    1. 25.3.1 Factoring
    2. 25.3.2 The Discrete Fourier Transform
    3. 25.3.3 Shor's Algorithm
    4. 25.3.4 Final Words
4. 25.4 Exercises

29. Appendix A Mathematica® Examples

    1. A.1 Getting Started with Mathematica
    2. A.2 Some Commands
    3. A.3 Examples for Chapter 2
    4. A.4 Examples for Chapter 3
    5. A.5 Examples for Chapter 5
    6. A.6 Examples for Chapter 6
    7. A.7 Examples for Chapter 9
    8. A.8 Examples for Chapter 10
    9. A.9 Examples for Chapter 12
    10. A.10 Examples for Chapter 17
    11. A.11 Examples for Chapter 18
    12. A.12 Examples for Chapter 21

30. Appendix B Maple® Examples

    1. B.1 Getting Started with Maple
    2. B.2 Some Commands
    3. B.3 Examples for Chapter 2
    4. B.4 Examples for Chapter 3
    5. B.5 Examples for Chapter 5
    6. B.6 Examples for Chapter 6
    7. B.7 Examples for Chapter 9
    8. B.8 Examples for Chapter 10
    9. B.9 Examples for Chapter 12
    10. B.10 Examples for Chapter 17
    11. B.11 Examples for Chapter 18
    12. B.12 Examples for Chapter 21

31. Appendix C MATLAB® Examples

1. C.1 Getting Started with MATLAB
2. C.2 Examples for Chapter 2
3. C.3 Examples for Chapter 3
4. C.4 Examples for Chapter 5
5. C.5 Examples for Chapter 6
6. C.6 Examples for Chapter 9
7. C.7 Examples for Chapter 10
8. C.8 Examples for Chapter 12
9. C.9 Examples for Chapter 17
10. C.10 Examples for Chapter 18
11. C.11 Examples for Chapter 21

32. Appendix D Sage Examples

    1. D.1 Computations for Chapter 2
    2. D.2 Computations for Chapter 3
    3. D.3 Computations for Chapter 5
    4. D.4 Computations for Chapter 6
    5. D.5 Computations for Chapter 9
    6. D.6 Computations for Chapter 10
    7. D.7 Computations for Chapter 12
    8. D.8 Computations for Chapter 17
    9. D.9 Computations for Chapter 18
    10. D.10 Computations for Chapter 21

33. Appendix E Answers and Hints for Selected Odd-Numbered Exercises
34. Appendix F Suggestions for Further Reading
35. Bibliography
36. Index

Landmarks

1. Frontmatter
2. Start of Content
3. backmatter

1. i
2. ii
3. iii
4. iv
5. v
6. vi
7. vii
8. viii
9. ix
10. x
11. xi
12. xii
13. xiii
14. xiv
15. 1
16. 2
17. 3
18. 4
19. 5
20. 6
21. 7
22. 8
23. 9
24. 10
25. 11

26. 12
27. 13
28. 14
29. 15
30. 16
31. 17
32. 18
33. 19
34. 20
35. 21
36. 22
37. 23
38. 24
39. 25
40. 26
41. 27
42. 28
43. 29
44. 30
45. 31
46. 32
47. 33
48. 34
49. 35
50. 36
51. 37
52. 38
53. 39
54. 40
55. 41
56. 42
57. 43

58. 44
59. 45
60. 46
61. 47
62. 48
63. 49
64. 50
65. 51
66. 52
67. 53
68. 54
69. 55
70. 56
71. 57
72. 58
73. 59
74. 60
75. 61
76. 62
77. 63
78. 64
79. 65
80. 66
81. 67
82. 68
83. 69
84. 70
85. 71
86. 72
87. 73
88. 74
89. 75

90. 76
91. 77
92. 78
93. 79
94. 80
95. 81
96. 82
97. 83
98. 84
99. 85
100. 86
101. 87
102. 88
103. 89
104. 90
105. 91
106. 92
107. 93
108. 94
109. 95
110. 96
111. 97
112. 98
113. 99
114. 100
115. 101
116. 102
117. 103
118. 104
119. 105
120. 106
121. 107

122. 108
123. 109
124. 110
125. 111
126. 112
127. 113
128. 114
129. 115
130. 116
131. 117
132. 118
133. 119
134. 120
135. 121
136. 122
137. 123
138. 124
139. 125
140. 126
141. 127
142. 128
143. 129
144. 130
145. 131
146. 132
147. 133
148. 134
149. 135
150. 136
151. 137
152. 138
153. 139

154. 140
155. 141
156. 142
157. 143
158. 144
159. 145
160. 146
161. 147
162. 148
163. 149
164. 150
165. 151
166. 152
167. 153
168. 154
169. 155
170. 156
171. 157
172. 158
173. 159
174. 160
175. 161
176. 162
177. 163
178. 164
179. 165
180. 166
181. 167
182. 168
183. 169
184. 170
185. 171

186. 172
187. 173
188. 174
189. 175
190. 176
191. 177
192. 178
193. 179
194. 180
195. 181
196. 182
197. 183
198. 184
199. 185
200. 186
201. 187
202. 188
203. 189
204. 190
205. 191
206. 192
207. 193
208. 194
209. 195
210. 196
211. 197
212. 198
213. 199
214. 200
215. 201
216. 202
217. 203

218. 204
219. 205
220. 206
221. 207
222. 208
223. 209
224. 210
225. 211
226. 212
227. 213
228. 214
229. 215
230. 216
231. 217
232. 218
233. 219
234. 220
235. 221
236. 222
237. 223
238. 224
239. 225
240. 226
241. 227
242. 228
243. 229
244. 230
245. 231
246. 232
247. 233
248. 234
249. 235

250. 236
251. 237
252. 238
253. 239
254. 240
255. 241
256. 242
257. 243
258. 244
259. 245
260. 246
261. 247
262. 248
263. 249
264. 250
265. 251
266. 252
267. 253
268. 254
269. 255
270. 256
271. 257
272. 258
273. 259
274. 260
275. 261
276. 262
277. 263
278. 264
279. 265
280. 266
281. 267

282. 268
283. 269
284. 270
285. 271
286. 272
287. 273
288. 274
289. 275
290. 276
291. 277
292. 278
293. 279
294. 280
295. 281
296. 282
297. 283
298. 284
299. 285
300. 286
301. 287
302. 288
303. 289
304. 290
305. 291
306. 292
307. 293
308. 294
309. 295
310. 296
311. 297
312. 298
313. 299

314. 300
315. 301
316. 302
317. 303
318. 304
319. 305
320. 306
321. 307
322. 308
323. 309
324. 310
325. 311
326. 312
327. 313
328. 314
329. 315
330. 316
331. 317
332. 318
333. 319
334. 320
335. 321
336. 322
337. 323
338. 324
339. 325
340. 326
341. 327
342. 328
343. 329
344. 330
345. 331

346. 332
347. 333
348. 334
349. 335
350. 336
351. 337
352. 338
353. 339
354. 340
355. 341
356. 342
357. 343
358. 344
359. 345
360. 346
361. 347
362. 348
363. 349
364. 350
365. 351
366. 352
367. 353
368. 354
369. 355
370. 356
371. 357
372. 358
373. 359
374. 360
375. 361
376. 362
377. 363

378. 364
379. 365
380. 366
381. 367
382. 368
383. 369
384. 370
385. 371
386. 372
387. 373
388. 374
389. 375
390. 376
391. 377
392. 378
393. 379
394. 380
395. 381
396. 382
397. 383
398. 384
399. 385
400. 386
401. 387
402. 388
403. 389
404. 390
405. 391
406. 392
407. 393
408. 394
409. 395

410. 396
411. 397
412. 398
413. 399
414. 400
415. 401
416. 402
417. 403
418. 404
419. 405
420. 406
421. 407
422. 408
423. 409
424. 410
425. 411
426. 412
427. 413
428. 414
429. 415
430. 416
431. 417
432. 418
433. 419
434. 420
435. 421
436. 422
437. 423
438. 424
439. 425
440. 426
441. 427

442. 428
443. 429
444. 430
445. 431
446. 432
447. 433
448. 434
449. 435
450. 436
451. 437
452. 438
453. 439
454. 440
455. 441
456. 442
457. 443
458. 444
459. 445
460. 446
461. 447
462. 448
463. 449
464. 450
465. 451
466. 452
467. 453
468. 454
469. 455
470. 456
471. 457
472. 458
473. 459

474. 460
475. 461
476. 462
477. 463
478. 464
479. 465
480. 466
481. 467
482. 468
483. 469
484. 470
485. 471
486. 472
487. 473
488. 474
489. 475
490. 476
491. 477
492. 478
493. 479
494. 480
495. 481
496. 482
497. 483
498. 484
499. 485
500. 486
501. 487
502. 488
503. 489
504. 490
505. 491

506. 492
507. 493
508. 494
509. 495
510. 496
511. 497
512. 498
513. 499
514. 500
515. 501
516. 502
517. 503
518. 504
519. 505
520. 506
521. 507
522. 508
523. 509
524. 510
525. 511
526. 512
527. 513
528. 514
529. 515
530. 516
531. 517
532. 518
533. 519
534. 520
535. 521
536. 522
537. 523

538. 524
539. 525
540. 526
541. 527
542. 528
543. 529
544. 530
545. 531
546. 532
547. 533
548. 534
549. 535
550. 536
551. 537
552. 538
553. 539
554. 540
555. 541
556. 542
557. 543
558. 544
559. 545
560. 546
561. 547
562. 548
563. 549
564. 550
565. 551
566. 552
567. 553
568. 554
569. 555

570. 556
571. 557
572. 558
573. 559
574. 560
575. 561
576. 562
577. 563
578. 564
579. 565
580. 566
581. 567
582. 568
583. 569
584. 570
585. 571
586. 572
587. 573
588. 574
589. 575
590. 576
591. 577
592. 578
593. 579
594. 580
595. 581
596. 582
597. 583
598. 584
599. 585
600. 586
601. 587

602. 588
603. 589
604. 590
605. 591
606. 592
607. 593
608. 594
609. 595
610. 596
611. 597
612. 598
613. 599
614. 600
615. 601
616. 602
617. 603
618. 604
619. 605
620. 606
621. 607
622. 608
623. 609
624. 610
625. 611
626. 612
627. 613
628. 614
629. 615
630. 616
631. 617
632. 618
633. 619

634. 620
635. 621

Long description

The illustration shows Alice is connected to encrypt with an arrow labeled as plaintext pointed towards encrypt. Encrypt is connected to decrypt with an arrow labeled as ciphertext. Decrypt is connected to Bob. Encrypt is connected with Encryption Key with an arrow pointed downward Encrypt. Decrypt is connected with Decryption Key with an arrow pointed downward Decrypt. The arrow between Encrypt and Decrypt is connected with Eve with an arrow pointed downward.

Long description

The illustration shows a box with three rows. In the rows, the letters are shown with their frequencies.

In the first row, a: .082, b: .015, c: .028, d: .043, e: .127, f: .022, g: .020, h: .061, i: .070, j:.002. In the second row, k: .008, l: .040, m: .024, n:.067, o: .075, p: .019, q: .001, r: .060, s: .063, t: .091.

In the third row, u: .028, v: .010, w: .023, x: .001, y: .020, z: .001.

Long description

The illustration shows a box. In the box, the letters are shown with their frequencies, e: .127, t: .091, a: .082, o: .075, i: .070, n: .067, s: .063, h: .061, r: .060.

Long description

The table shows counting diagrams with 9 rows labeled as W, B, R, S, I, V, A, P, N and 9 columns labeled as W, B, R, S, I, V, A, P, N. The table shows various corresponding values for each row and each column.

Long description

The table shows 5 versus 5 matrix table with row labeled as A, D, F, G, X and columns labeled as A, D, F, G, X. The table shows various corresponding values for each column and each row.

Long description

The table shows a matrix table with 5 columns labeled as R, H, E, I, N. The table shows various corresponding various values for each column.

Long description

The table shows a matrix table with 5 columns labeled as E, H, I, N, R. The table shows various corresponding values for each column.

Long description

The illustration shows four rectangular boxes labeled as R, L, M, N respectively placed vertically parallel to each other. The illustration shows a square box labeled as S beyond the four boxes. S is connected to N, N is connected to M, M is connected to L, L is connected to R with an arrow and vice versa. Beyond S, a rectangular box labeled as K, keyboard is connected to S with an arrow at the very lower position and a bar with four bulbs labeled as glow lamps is connected to S with an arrow at the very upper position.

Long description

A table shows 6 rows and 6 columns for addition mod 6. The table contains various values for each rows and columns.

Long description

A table shows 6 rows and 6 columns for multiplication mod 6. The table contains various values for each rows and columns.

Long description

The table shows various symbols and their decimal and binary number in five numbers of rows. The first row contains the symbol exclamation : 33 : 0100001, double inverted : 34 : 0100010, hash : 35 : 0100011, Dollar : 36 : 0100100, percentage : 37 : 0100101, and : 38 : 0100110, single inverted : 39 : 0100111. The second row contains the symbols open bracket : 40 : 0101000, closed bracket : 41 : 0101001, asterisk : 42 : 0101010, plus : 43 : 0101011, coma : 44 : 0101100, minus : 45 : 0101101, dot : 46 : 0101110, slash: 47 : 0101111. The third row contains 0 : 48 : 0110000, 1 : 49 : 011000, 2 : 50 : 0110010, 3 : 51 :

0110011, 4 : 52 : 0110100, 5 : 53 : 0110101, 6 : 54 : 011010, 7 : 55 : 0110111. The fourth row contains 8 : 56 : 0111000, 9 : 57 : 0111001, colon : 58 : 0111010, semi colon : 59 : 0111011, less than : 60 : 0111100, equal : 61 : 0111101, greater than : 62 : 0111110, question mark : 63 : 0111111. The last row contains the symbols at the rate : 64 : 1000000, A : 65 : 1000001, B : 66 : 1000010, C : 67 : 1000011, D : 68 : 1000100, E : 69 : 1000100, F : 70 : 1000110, G : 71 : 1000111.

Long description

The block diagram of a stream cipher encryption shows an arrow pointed to x subscript n plus 2, x subscript n plus 2 is connected to x subscript n plus 1 with an arrow, and then x subscript n plus 1 is connected to x subscript n with an arrow which further connects to an adder. Then, an arrow points towards p subscript n plus 2 which is connected to p subscript n plus 1 with an arrow, and then p subscript n plus 1 is points to p subscript n with an arrow which further connects to an adder. The output of the adder is fed to c subscript n, from which an output is obtained.

Long description

The image of a linear feedback shift register satisfying x subscript n plus 3 equals x subscript n plus 1 plus x subscript n shows that x subscript n plus 2 is connected to x subscript n plus 1 with an arrow which is further connected to x subscript n with an arrow. x subscript n plus 2 and x subscript n is further fed to an adder and the output of the adder is again fed to x subscript n plus 2. x subscript n and plaintext are fed to another adder from which the output ciphertext is obtained.

Long description

The first block contains P subscript 1 which represents the first plain text which is then connected to block E subscript K which represents encryption function. Block C subscript 0 which represents initialization vector is connected to block P subscript 1 and E subscript K through a connector. Block E subscript K is then connected to block C subscript 1 which represents the first ciphertext. Block that contains P subscript 2 which represents second plain text is connected to E subscript K through a connector. Block C subscript 1 is connected to the connector that connects block P subscript 2 and block

E subscript K. Again, the block E Subscript K connects the block C subscript 2 which represents the second ciphertext from where the arrow connects to dot-dot-dot.

Long description

The illustration shows eleven blocks connected to each other. The first block labeled as X subscript 1 connects to the second block E subscript which again, connects to the block that contains O subscript 1 in a register that denotes 8 leftmost bits that connect with a connector P subscript 1 that shows 8 bits and C subscript 1 is connected to the connector. Again, C subscript 1 is connected to register that contains C subscript 1 present in rightmost side of the block X subscript 2 that connects the block X subscript 2 and shifts the rightmost register labeled as drop to the leftmost side that is represented by

downward arrows which connects to the block E subscript K which again connects to the block that contains O subscript 2 in a register that denotes 8 leftmost bits that connects with a connector P subscript 2 that shows 8 bits and C subscript 2 is connected to the connector. Again C subscript 2 is connected to register that contains C subscript 2 present in rightmost side of the block X subscript 2 that connects the block X subscript 3 and shifts the rightmost register labeled as drop to the leftmost side that is represented by downward arrows which connects to the block E subscript K which again connects to the block that contains O subscript 3 in a register that denotes 8 leftmost bits that connects with a

connector P subscript 2 that shows 8 bits and C subscript 3 is connected to the connector which is further connected to dot-dot-dot. The first block X subscript 1 is connected to the next X subscript 1 block that contains O subscript 1 and X subscript 2 block is connected to the next X subscript 2 block that contains O subscript 2.

Long description

The illustration shows eleven blocks connected to each other. The first block labeled as X subscript 1 connects to the second block E subscript which again connects to the block that contains O subscript 1 in a register that denotes 8 leftmost bits that connect with a connector P subscript 1 that shows 8 bits and C subscript 1 is connected to the connector. Again, O subscript 1 is connected to register that contains O subscript 1 present in rightmost side of the block X subscript 1 that connects the block X subscript 2 and shifts the rightmost register labeled as drop to the leftmost side that is represented by

downward arrows which connects to the block E subscript K which again connects to the block that contains O subscript 2 in a register that denotes 8 leftmost bits that connects with a connector P subscript 2 that shows 8 bits and C subscript 2 is connected to the connector. Again O subscript 2 is connected to register that contains O subscript 2 present in rightmost side of the block X subscript 2 that connects the block X subscript 3 and shifts the rightmost register labeled as drop to the leftmost side that is represented by downward arrows which connects to the block E subscript K which again connects to the block that contains O subscript 3 in a register that denotes 8 leftmost bits that connects with a

connector P subscript 2 that shows 8 bits and C subscript 3 is connected to the connector which is further connected to dot-dot-dot. The first block X subscript 1 is connected to the next X subscript 1 block that contains O subscript 1 and X subscript 2 block is connected to the next X subscript 2 block that contains O subscript 2.

Long description

The illustration shows nine blocks that are separated. The first block labeled as X subscript 1 connects to the second block E subscript which again connects to the block that contains O subscript 1 in a register that denotes 8 leftmost bits that connect with a connector P subscript 1 that shows 8 bits and C subscript 1 is connected to the connector. Block X subscript 2 equals X subscript 1 plus 1 connects to the block E subscript K which again connects to the block that contains O subscript 2 in a register that denotes 8 leftmost bits that connect with a connector P subscript 2 that shows 8 bits and C subscript 2 is

connected to the connector. Block X subscript 3 equals X subscript 2 plus 1 connects to the block E subscript K which again connects to the block that contains O subscript 3 in a register that denotes 8 leftmost bits that connect with a connector P subscript 2 that shows 8 bits and C subscript 3 is connected to the connector which is further connected to dot-dot-dot.

Long description

The illustration shows two inputs K subscript i and R subscript i minus l fed to a feedback which is labeled as f. Output of feedback is fed to an adder which is denoted as XOR. A output R subscript i is obtain from the adder. Another input L subscript I minus l is fed to the adder. An arrow is shown from R subscript i minus l which is labeled as L subscript i.

Long description

The illustration shows six numbers, 1, 2, 3, 4, 5, 6 on the upper row and eight numbers 1, 2, 4, 3, 4, 3, 5, 6 on lower row respectively. Several arrows are shown between 1 to 1, 2 to 2, 4 to 4, 3 to 3, 4 to 4, 3 to 3, 5 to 5, 6 to 6 from upper row to bottom row.

Long description

Flow chart shows an input R subscript i minus l which is fed to E left parenthesis R subscript i minus l right parenthesis. Output of E left parenthesis R subscript i minus l right parenthesis is fed to an adder. Adder output is subdivided into two parts 4 bits and 4 bits. Output of 4bits and 4 bits are connected to S subscript 1 and S subscript 2 respectively. Output of the S subscript 1 and S subscript 2 is connected to f left parenthesis R subscript i minus l, K subscript i right parenthesis. On the right side, another input is fed to an adder which is labeled as K subscript i.

Long description

Table shows two columns, first four bits along with their frequencies, the values are labeled as 0000: 12, 1000: 33, 0001: 7, 1001: 40, 0010: 8, 1010: 35, 0011: 15, 1011: 35, 0100: 4, 1100: 59, 0101: 3, 1101: 32, 0110: 4, 1110: 28, 0111: 6 and 1111: 39.

Another table shows two columns, last four bits along with their frequencies, the values are labeled as 0000: 14, 1000: 8, 0001: 6, 1001: 16, 0010: 42, 1010: 8, 0011: 10, 1011: 18, 0100: 27, 1100: 8, 0101: 10, 1101: 23, 0110: 8, 1110: 6, 0111: 11, and 1111: 17.

Long description

An input Plaintext is fed to IP. The output of IP is subdivided in two parts L subscript 0 and R subscript 0 respectively. The output of L subscript 0 is connected to an adder whereas the output of R subscript 0 is connected to the adder through a feedback f, where an input K subscript 1 is fed. Output of the adder is connected to R subscript 1 and same of R subscript 0 is connected to L subscript 1. The output of L subscript 1 is connected to an adder whereas the output of R subscript 1 is connected to the adder through a feedback f, where an input K subscript 2 is fed. Output of the adder is connected to R subscript 2 and

same of R subscript 1 is connected to L subscript 2. L subscript 2 and R subscript 2 are further connected to L subscript 15 and R subscript 15 respectively. The output of L subscript 15 is connected to an adder whereas the output of R subscript 15 is connected to the adder through a feedback f, where an input K subscript 16 is fed. Output of the adder is connected to R subscript 16 and same of R subscript 15 is connected to L subscript 16. The outputs obtained from R subscript 16 and L subscript 16 are fed to IP superscript negative 1 which is further fed to ciphertext.

Long description

The values are labeled as follows:

Row 1: 58 50 42 34 26 18 10 2 60 52 44 36 28 20 12 4

Row 2: 62 54 46 38 30 22 14 6 64 56 48 40 32 24 16 8

Row 3: 57 49 41 33 25 17 9 1 59 51 43 35 27 19 11 3

Row 4: 61 53 45 37 29 21 13 5 63 55 47 39 31 23 15 7

Long description

The values are labeled as follows:

Row 1: 32 1 2 3 4 5 4 5 6 7 8 9

Row 2: 8 9 10 11 12 13 12 13 14 15 16 17

Row 3: 16 17 18 19 20 21 20 21 22 23 24 25

Row 4: 24 25 26 27 28 29 28 29 30 31 32 1

Long description

The values are labeled as follows:

Row 1: 16 7 20 21 29 12 28 17 1 15 23 26 5 18 31 10

Row 2: 2 8 24 14 32 27 3 9 19 13 30 6 22 11 4 25

Long description

A flow chart starts with an input R subscript i minus 1 which is fed to Expander. The output of expander is fed to E left parenthesis R i minus 1 right parenthesis. The output of E left parenthesis R i minus 1 right parenthesis and another input K subscript i are fed to an adder. The output of the adder is subdivided into eight 6 bits data which are labeled as B subscript 1, B subscript 2, B subscript 3, B subscript 4, B subscript 5, B subscript 6, B subscript 7 and B subscript 8 respectively. The outputs of B subscript 1, B subscript 2, B subscript 3, B subscript 4, B subscript 5, B subscript 6, B subscript 7 and B

subscript 8 respectively are fed to S subscript 1, S subscript 2, S subscript 3, S subscript 4, S subscript 5, S subscript 6, S subscript 7 and S subscript 8 respectively. Again the outputs of S subscript 1, S subscript 2, S subscript 3, S subscript 4, S subscript 5, S subscript 6, S subscript 7 and S subscript 8 respectively are fed to eight 4 bits data which are labeled as C subscript 1, C subscript 2, C subscript 3, C subscript 4, C subscript 5, C subscript 6, C subscript 7 and C subscript 8 respectively. All the outputs are fed to permutation. The output of permutation is fed to f left parenthesis R subscript i minus 1 comma K i right parenthesis.

Long description

The values are labeled as follows:

Row 1: 57 49 41 33 25 17 9 1 58 50 42 34 26 18

Row 2: 10 2 59 51 43 35 27 19 11 3 60 52 44 36

Row 3: 63 55 47 39 31 23 15 7 62 54 46 38 30 22

Row 4: 14 6 61 53 45 37 29 21 13 5 28 20 12 4

Long description

The values are labeled as follows:

Round: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Shift: 1 1 2 2 2 2 2 2 1 2 2 2 2 2 2 1

Long description

A table shows information of 48 bits chosen from the 56-bit string C subscript i D subscript i and the output are K subscript i. The table shows four rows and twelve columns.

Row 1: 14, 17, 11, 24, 1, 5, 3, 28, 15, 6, 21 and 10.

Row 2: 23, 19, 12, 4, 26, 8, 16, 7, 27, 20, 13 and 2.

Row 3: 41, 52, 31, 37, 47, 55, 30, 40, 51, 45, 33 and 48.

Row 4: 44, 49, 39, 56, 34, 53, 46, 42,

50, 36, 29 and 32.

Long description

A table shows information of S-Boxes. The table divided into eight parts depicted as S-box 1, S-box 2, S-box 3, S-box 4, S-box 5, S-box 6, S-box 7 and S-box 8 where each table shows four rows and 16 columns with several numbers.

Long description

A box labeled as plaintext is linked with the box AddRoundKey, The box AddRoundKey is labeled as W left parenthesis 0 right parenthesis, W left parenthesis 1 right parenthesis, W left parenthesis 2 right parenthesis, W left parenthesis 3 right parenthesis with a left inward arrow. Below the box AddRoundKey, a block with four boxes labeled as Round 1. In the block the first box is SubBytes which is linked with the below box labeled as ShiftRows. Again the ShiftRows is linked with MixColumns and lastly the MixColumns box is linked with AddRoundKey which is labeled as W

left parenthesis 4 right parenthesis, W left parenthesis 5 right parenthesis, W left parenthesis 6 right parenthesis, W left parenthesis 7 right parenthesis with a left inward arrow. Again a block with four boxes labeled as Round 9, starting with the first box labeled as SubBytes which is linked with ShiftRows. Again ShiftRows is linked with MixColumns and lastly the MixColumns box is linked with AddRound Key which is labeled as W left parenthesis 36 right parenthesis, W left parenthesis 37 right parenthesis, W left parenthesis 38 right parenthesis, W left parenthesis 39 right parenthesis with a left inward arrow. Again a block with three boxes labeled as Round 10, starting with the first box labeled as SubBytes which is linked with

ShiftRows. Again ShiftRows is linked with AddRound Key which is labeled as W left parenthesis 40 right parenthesis, W left parenthesis 41 right parenthesis, W left parenthesis 42 right parenthesis, W left parenthesis 43 right parenthesis with a left inward arrow. Below the block, a box is linked labeled as Ciphertext.

Long description

A table shows the information of Rijndael Encryption. The data listed for:

Line 1: ARK, using the 0^{th} round key.

Line 2: Nine rounds of SB, SR, MC, ARK, using round keys 1 to 9.

Line 3: A final round: SB, SR, ARK, using the 10^{th} round key.

Long description

A table shows S-Box for Rijndael that shows various numerical values. The values are listed as

99 124 119 123 242 107 111 197 48 1 103 43 254 215 171 118

202 130 201 125 250 89 71 240 173 212 162 175 156 164 114 192

183 253 147 38 54 63 247 204 52 165 229 241 113 216 49 21

4 199 35 195 24 150 5 154 7 18 128 226 235 39 178 117

9 131 44 26 27 110 90 160 82 59 214
179 41 227 47 132

83 209 0 237 32 252 177 91 106 203
190 57 74 76 88 207

208 239 170 251 67 77 51 133 69 249 2
127 80 60 159 168

81 163 64 143 146 157 56 245 188 182
218 33 16 255 243 210

205 12 19 236 95 151 68 23 196 167
126 61 100 93 25 115

96 129 79 220 34 42 144 136 70 238
184 20 222 94 11 219

224 50 58 10 73 6 36 92 194 211 172 98
145 149 228 121

231 200 55 109 141 213 78 169 108 86
244 234 101 122 174 8

186 120 37 46 28 166 180 198 232 221
116 31 75 189 139 138

112 62 181 102 72 3 246 14 97 53 87
185 134 193 29 158

225 248 152 17 105 217 142 148 155 30
135 233 206 85 40 223

140 161 137 13 191 230 66 104 65 153
45 15 176 84 187 22

Long description

A table shows the information of Rijndael Decryption. The data listed for

line 1: ARK, using the 10th round key.

Line 2: Nine rounds of ISB, ISR, IMC, IARK, using round keys 9 to 1.

Line 3: A final round: ISB, ISR, ARK, using the 0th round key.

Long description

A table shows information of The RSA Algorithm. The data listed for

Line 1: Bob chooses secret primes p and q and computes n equal pq.

Line 2: Bob chooses e with gcd left parenthesis e comma left parenthesis p minus 1 right parenthesis left parenthesis q minus 1 double right parenthesis equals 1.

Line 3: Bob computers d with de-triple equal 1 left parenthesis mod left parenthesis p minus 1 right parenthesis left parenthesis q minus 1 double right

parenthesis.

Line 4: Bob makes n and e public comma and keeps p comma q comma d secret.

Line 5: Alice encrypts m as c triple equal m superscript e left parenthesis mod n right parenthesis and sends c to Bob.

Line 6: Bob decrypts by computing m triple equal c superscript d left parenthesis mod n.

Long description

A table shows information of RSA Encryption Exponents. The data listed for e and percentage as follows:

Column 1: 65537, 17, 41, 3, 19, 25, 5, 7, 11, 257 and others.

Column 2: 95.4933 percent, 3.1035 percent, 0.4574 percent, 0.3578 percent, 0.1506 percent, 0.1339 percent, 0.1111 percent, 0.0596 percent, 0.0313 percent, 0.0241 percent and 0.0774 percent.

Long description

A table shows information of Factorization Records. The data listed for year and number of digits as follows:

Column 1: 1964, 1974, 1984, 1994, 1999, 2003, 2005 and 2009.

Column 2: 20, 45, 71, 129, 155, 174, 200 and 232.

Long description

The long message is an arbitrary message like …… 01101001… which pass through the hash function and produces the output message digest of fixed length 256 bit- 11…10.

Long description

Initially, a value IV is fed as input in the first block f. A two-bit string M sub 0 is first fed into the block. The blocks are then fed one-by-one into f. The message M goes from M sub 0 to up to M sub left parenthesis n-1right parenthesis. The final output is the hash value H left parenthesis M right parenthesis.

Long description

r is taken as the rate and c is taken as the capacity. First a block 0 0 is introduced. First 0 is the rate r and second o is the capacity c. Rate r occupies maximum part of the block. Then the message signal M subscript 0 is transferred to block f from where the message is transferred back to rate and capacity block and so on till message signal M subscript n minus 1. This is the absorbing part. The diagram is then separated by a dotted line after which the squeezing part starts. Here the outputs Z subscript 0 and so on are squeezed out of the rate and capacity and f blocks.

Long description

An illustration shows three blocks labeled as Data Record k minus 1, Data Record k, and Data Record k + 1 and each block contains a data block, h left parenthesis right parenthesis, and a pointer. A vertical box with h left parenthesis right parenthesis and a pointer is shown at the upper top right. The vertical box is linked with the third block with the help of an arrow. Again the third block is linked with the second box and also the second box is linked with the first block with the help of an arrow.

Long description

An illustration shows a tree diagram. The first block of the tree diagram is h left parenthesis right parenthesis. The first block is linked with the second block with a downward arrow and the second block has h left parenthesis right parenthesis h left parenthesis right parenthesis. The second block is divided into two child blocks labeled as h left parenthesis right parenthesis h left parenthesis right parenthesis and h left parenthesis right parenthesis h left parenthesis right parenthesis. Again the block h left parenthesis right parenthesis h left parenthesis right parenthesis is subdivided into two child blocks labeled

as h left parenthesis right parenthesis h left parenthesis right parenthesis and h left parenthesis right parenthesis h left parenthesis right parenthesis. Also, the block h left parenthesis right parenthesis h left parenthesis right parenthesis is subdivided into two child blocks labeled as h left parenthesis right parenthesis h left parenthesis right parenthesis and h left parenthesis right parenthesis h left parenthesis right parenthesis. The block h left parenthesis right parenthesis h left parenthesis right parenthesis is subdivided into two child blocks labeled as R0 and R1. The block h left parenthesis right parenthesis h left parenthesis right parenthesis is subdivided into two child blocks labeled as R2 and R3. The block h left

parenthesis right parenthesis h left parenthesis right parenthesis is subdivided into two child blocks labeled as R4 and R5. The block h left parenthesis right parenthesis h left parenthesis right parenthesis is subdivided into two child blocks labeled as R6 and R7.

Long description

The illustration shows a basic Kerberos model with the participants Cliff, Serge, Trent and Grant, where Cliff is at the center which is linked to Trent, Grant and Serge. Cliff sends a message to Trent indicated by an arrow numbered 1. Trent replies the message to Cliff again indicated by an arrow numbered 2. Cliff sends a message to Grant indicated by an arrow numbered 3. Grant replies the message to Cliff again indicated by an arrow numbered 4. Cliff sends a final message to Serge indicated by an arrow numbered 5.

Long description

The illustration shows a block diagram of a Certification Hierarchy. The block “CA” abbreviated for Certification Authority, is characterized as Client, Client and “RA” abbreviated for Registration Authorities. The block “RA” is again sub-divided into three Clients.

Long description

The screenshot shows a CA’s Certificate; General.

Line 1: This certificate has been verified for the following uses, colon

Line 2: Email Signer Certificate

Line 3: Email Recipient Certificate

Line 4: Status Responder Certificate

Line 5: Issued to, colon

Line 6: Organization left parenthesis O right parenthesis, colon, VeriSign, Inc

Line 7: Organizational Unit left parenthesis OU right parenthesis, colon, Class 1 Public Primary Certification Authority dash G2

Line 8: Serial Number

Line 9: Issued by, colon

Line 10: Organization left parenthesis O right parenthesis, colon, VeriSign, Inc

Line 11: Organizational Unit left parenthesis OU right parenthesis, colon, Class 1 Public Primary Certification Authority dash G2

Line 12: Validity, colon

Line 13: Issues on, colon, 05 forward

slash 17 forward slash 98

Line 14: Expires on, colon, 05 forward slash 18 forward slash 98

Line 15: Fingerprints, colon

Line 16: SHA1 Fingerprint, colon

Line 17: MD5 Fingerprint, colon

Long description

The screenshot shows a CA’s Certificate; Details.

Line 1: Certificate Hierarchy

Line 2: Verisign Class 1 Public Primary Certification Authority dash G2

Line 3: Certificate Fields

Line 4: Verisign Class 1 Public Primary Certification Authority dash G2

Line 5: Certificate

Line 6: Version, colon, Version 1

Line 7: Serial Number, colon

Line 8: Certificate Signature Algorithms, colon, PKCS hashtag 1 SHA dash 1 With RSA Encryption

Line 9: Issuer, colon, OU equals VeriSign Trust Network

Line 14: Validity

Line 15: Not before, colon, 05 forward slash 17 forward slash 98

Line 16: Not after, colon, 05 forward slash 18 forward slash 98

Line 17: Subject, colon, OU equals VeriSign Trust Network

Line 22: Subject Public Key Info, colon, PKCS hashtag 1 RSA Encryption

Line 23: Subject’s Public Key, colon

Line 24: A table of 9 rows and 16 columns is shown

Line 25: Certificate Signature Algorithm, colon, PKCS hashtag 1 SHA dash 1 With RSA Encryption

Line 26: Certificate Signature Name, colon

Line 27: A table of 8 rows and 16 columns is shown.

Long description

The screenshot shows a Clients Certificate.

Line 1: Certificate Hierarchy

Line 2: Verisign Class 3 Public Primary CA

Line 3: Certificate Fields

Line 4: Verisign Class 3 Public Primary Certification Authority

Line 5: Certificate

Line 6: Version, colon, Version 3

Line 7: Serial Number, colon

Line 8: Certificate Signature Algorithms, colon, md5RSA

Line 9: Issuer, colon, OU equals www.verisign.com forward slash CPS Incorp.

Line 14: Validity

Line 15: Not before, colon, Sunday, September 21, 2003

Line 16: Not after, colon, Wednesday, September 21, 2005

Line 17: Subject, colon, CN equals online.wellsfargo.com

Line 22: Subject Public Key Info, colon, PKCS hashtag 1 RSA Encryption

Line 23: Subject’s Public Key, colon, 30 81 89 02 81 81 00 a9

Line 24: Basic Constraints, colon, Subject Type equals End Entity, Path Length Constraint equals None

Line 25: Subject's Key Usage, colon, Digital Signature, Key Encipherment left parenthesis AO right parenthesis

Line 26: CRL Distribution Points, colon

Line 27: Certificate Signature Algorithm, colon, MD5 With RSA Encryption

Line 28: Certificate Signature Value, colon

Long description

A table represents Block ID colon 76 Create Coins. The table contains 4 rows and 3 columns. The Block ID colon 76 Create Coins is divided into three columns labeled as Trans ID with values 0, 1 and 2, Value with values 5, 2 and 10 and Recipient with values PK subscript BB, PK subscript Alice and PK subscript Bob.

Long description

A table represents Block ID colon 77 Consume Coins colon 41 left parentheses 2 right parentheses, 16 left parentheses 0 right parentheses, 31 left parentheses 1 right parentheses, Create Coins. The table contains 6 rows and 3 columns. The Block ID colon 77 Consume Coins colon 41 left parentheses 2 right parentheses, 16 left parentheses 0 right parentheses, 31 left parentheses 1 right parentheses, Create Coins is divided into three columns labeled as Trans ID with values 0, 1, 2 and 3, Value with values 15, 5, 4 and 11 and Recipient with values PK subscript Sarah Store, PK subscript Alice, PK

subscript Bob and PK subscript Charles.

Long description

The block diagram of Bit coin’s block chain and a Merkle tree of transactions three blocks in adjacent position. Each block has four lines of codes. The third block connects to the second and the second block connects to the first.

Line 1: prev underscore hash tag, colon, h left parenthesis right parenthesis

Line 2: timestamp

Line 3: merkleroot, colon, h left parenthesis right parenthesis

Line 4: nonce

Another block is placed just below the second block, which is comprised of one sub block, with a line: h left parenthesis right parenthesis, indented, h left parenthesis right parenthesis. Below the first, there are two other sub-blocks with lines in each block as: h left parenthesis right parenthesis, indented, h left parenthesis right parenthesis, connected with two arrows with the first block. The above two sub-blocks are connected to four sub-blocks represented as TX.

Long description

The box is having a uniform thickness highlighted with black. Another solid rectangle is present inside the tunnel and below it a door is made.

Long description

Illustration shows the Tunnel Used in the Zero-Knowledge Protocol. The tunnel is represented by a rectangular box with an opening at the top. Inside it, another tunnel is made by rectangular box with an opening at the top. Both the boxes are having a uniform thickness highlighted with black. Another solid rectangle is present inside the tunnel and a Central Chamber is presented below it.

Long description

A schematic diagram of Huffman Encoding, showing outputs a, b, c and d, arranged vertically. Output c corresponding to 0.1 and d corresponding to 0.1 gives 0.2, with two choices 0 and 1. Again, 0.2 and b corresponding 0.3 gives 0.5, with two choices 0 and 1. Further, 0.5 and a corresponding 0.5 gives 1, with two choices 0 and 1.

Long description

The illustration shows Shannon's Experiment on the Entropy of English. There are six sentences “there is no reverse”, “on a motorcycle a”, “friend of mine found”, “this out rather”, “dramatically the” and “other day”. Below each letter, information obtained is displayed.

Long description

The x-axis ranges from negative 1 to 3, in increments of 1 and the y-axis ranges from negative 4 to 4, increments of 2. The first curve begins at the first quadrant, passes decreasing through point 1 of the x-axis, then ends decreasing at the fourth quadrant making a peak point at point 1 of the x-axis.

Long description

The x-axis ranges from negative 4 to 8, in increments of 2 and the y-axis ranges from negative 20 to 20, in increments of 10. The curve starts from the first quadrant passing through the point 10 of the y-axis, then passes through the second quadrant decreasing through the point negative 4 of the x-axis intersecting the point negative 10 and ends decreasing at the fourth quadrant.

Long description

The graph shows a curve, a straight line and another doted straight line that is parallel to the y-axis and it is plotted in the first and second quadrant. The curve begins at the first quadrant passes decreasing through the y-axis, enters the second quadrant passes through the x-axis, enters the third quadrant passes through the y-axis and ends decreasing at the fourth quadrant.

Long description

The illustration shows a straight line segment and two vectors. The first vector labeled as v subscript 1 lies along the line segment pointing toward the right direction. The second vector labeled as v subscript 2 starts from the beginning point of the first vector, pointing upwards and slightly deflexed towards right from the angle 90 degrees. The second line lies between two dotted lines perpendicular to the line segment.

Long description

The illustration shows a straight line segment and two vectors. The first vector labeled as v subscript 1 lies along the line segment pointing toward the right direction. The second vector labeled as v subscript 2 starts from the beginning point of the first vector, pointing upwards and deflexed at angle 60 degrees towards the right. The second vector lies between two dotted lines perpendicular to the line segment and the vector crosses the perpendicular line at the very right.

Long description

The illustration shows an input message which is fed to the encoder, encoder output is fed into the noisy channel through codewords. Noisy channel output is fed to decoder and decoder output is fed to message.

Long description

The x-axis ranges from 0.1 to 0.5, in increments of 0.1 and the y-axis labeled as code rate ranges from 0.2 to 1, in increments of 0.2. The graph shows a non-linear decreasing curve starting from point 1 of the y-axis and ends at point 0.5 on the x-axis.

Long description

The illustration shows a white rectangular bar labeled as light source with a vertical edge at the right end. A black rectangular bar with a vertical edge labeled as Polaroid A at the right end which is inclined to a grey rectangular bar with a vertical black edge is shown. An arrow labeled as light pointing towards right is shown below the illustration.

Long description

The illustration shows a white rectangular bar labeled as light source with a vertical edge at the right end. A black rectangular bar with a vertical edge labeled as Polaroid A at the right end which is inclined to a grey rectangular bar with a vertical edge labeled as Polaroid C and a black vertical edge is are shown. An arrow labeled as light pointing towards right is shown below the illustration.

Long description

The illustration shows a white rectangular bar labeled as light source with a vertical edge at the right end. A black rectangular bar with a vertical edge labeled as Polaroid A at the right end which is inclined to a grey rectangular bar with a vertical edge labeled as Polaroid B is shown. The Polaroid B is inclined with a gray rectangular bar with a vertical bar labeled as Polaroid C which is inclined with a grey rectangular bar with a vertical black edge is shown. An arrow labeled as light pointing towards right is shown below the illustration.

Long description

The image shows a graph of y versus x. The x axis ranges from 0 to 7, in increments of 1 and the y-axis ranges from 0 to 1, in increments of 0.2. Following coordinates (0, 0.15), (1, 0.15), (2, 0.35), (3, 0.85), (4, 0.35), (5, 0.85), (6, 0.35) and (7, 0.15) are marked.

Long description

The image shows a graph of y versus x. The x axis ranges from 0 to 14, in increments of 2 and the y-axis ranges from 0 to 1, in increments of 0.2. Following coordinates (0, 1), (1, 0.2), (2, 0.3), (3, 0.9), (4, 0), (5, 0.2), (6, 0.65), (7, 0.3), (8, 0), (9, 0.3), (10, 0.65), (11, 0.2), (12, 0), (13, 0.9), (14, 0.25) and (15, 0.2) are marked.

Long description

The image shows a graph of y versus x. The x axis ranges from 0 to 500, in increments of 100 and the y-axis ranges from 0 to 0.2, in increments of 0.05. The graph starts from y equals 0.01, is comprised of four peaks with an altitude of y equals 0.02. The graph is comprised of numerous dots, whose density decreases towards the peaks.

Long description

A graph ranges negative 4 to 4 in increment of 2 in vertical axis and negative 1 to 3 in increment of 1 in horizontal axis. The graph plots an oval shaped curve at point 0 on vertical axis. Another elliptic curve is plotted that starts from 3 on horizontal axis, vertex lies at (1, 0) and extends to (3, 5).

Long description

A y versus x graph ranges from negative 4 to 4 in increment of 2 in y axis and negative 1 to 3 in increment of 1 in x axis. The graph plots an oval shaped curve whose vertex lies at the origin and (negative 1, 0). Another elliptic curve is plotted which is symmetric about the x axis and vertex lies at (1, 0).

Long description

A graph ranges negative 5 to 5 in increment of 1 in vertical axis and negative 1 to 3 in increment of 0.5 in horizontal axis. The graph plots an oval shaped curve at point 0 on vertical axis. Another elliptic curve is plotted that starts from 3 on horizontal axis, vertex lies at (1, 0) and extends to (3, 5). An equation is shown above the graph depicting y subscript 2 minus x times left parenthesis x minus 1 right parenthesis times left parenthesis x plus 1 right parenthesis equals 0.